The iOS Apprentice 2

Checklists (Part 1)

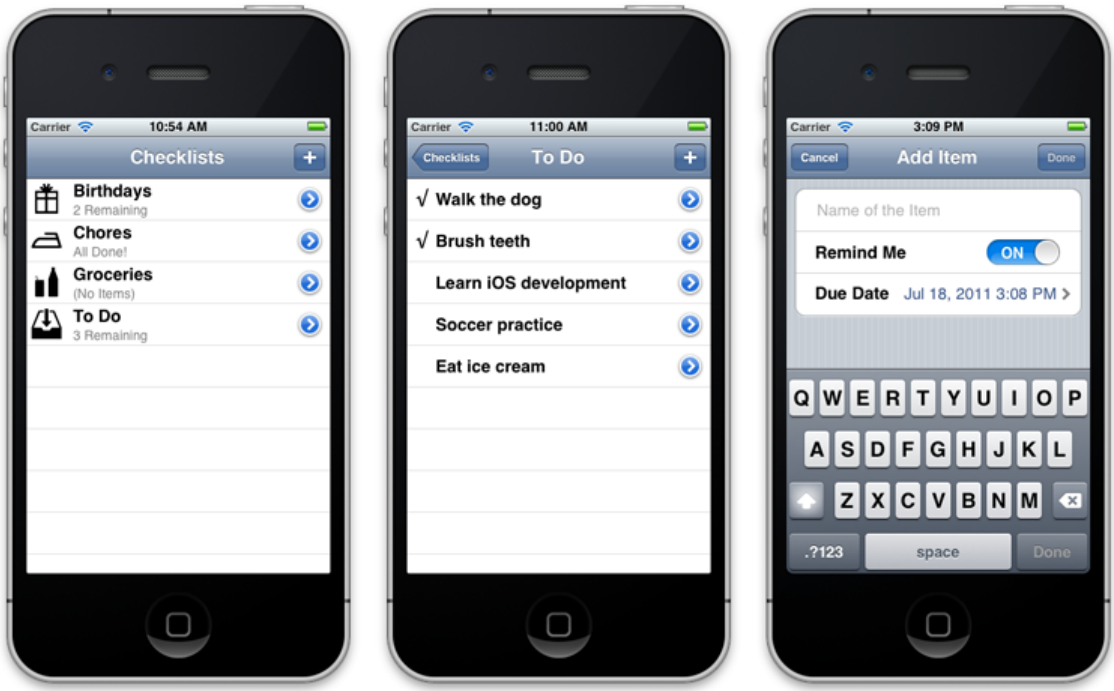
By Matthijs Hollemans

Version 1.4

To-do list apps are one of the most popular types of app on the App Store, second only to fart apps. Apple even included their own Reminders app with iOS 5 (but fortunately no built-in fart app). Building a to-do list app is somewhat of a rite of passage for budding iOS developers, so it makes sense that we create one as well.

Our own to-do list app, Checklists, will look like this when we're finished:

The finished Checklists app



The app lets you organize to-do items into lists and then check off these items once you're done with them. You can set a reminder on a to-do item that will make the iPhone pop up an alert on the due date, even when the app isn't running.

As far as to-do list apps go, Checklists is very basic, but don't let that fool you. Even a simple app such as this already has six different screens and a lot of complexity behind the scenes.

Table views and navigation controllers

This tutorial will introduce you to two of the most commonly used UI (user interface) elements in iOS apps: the *table view* and the *navigation controller*.

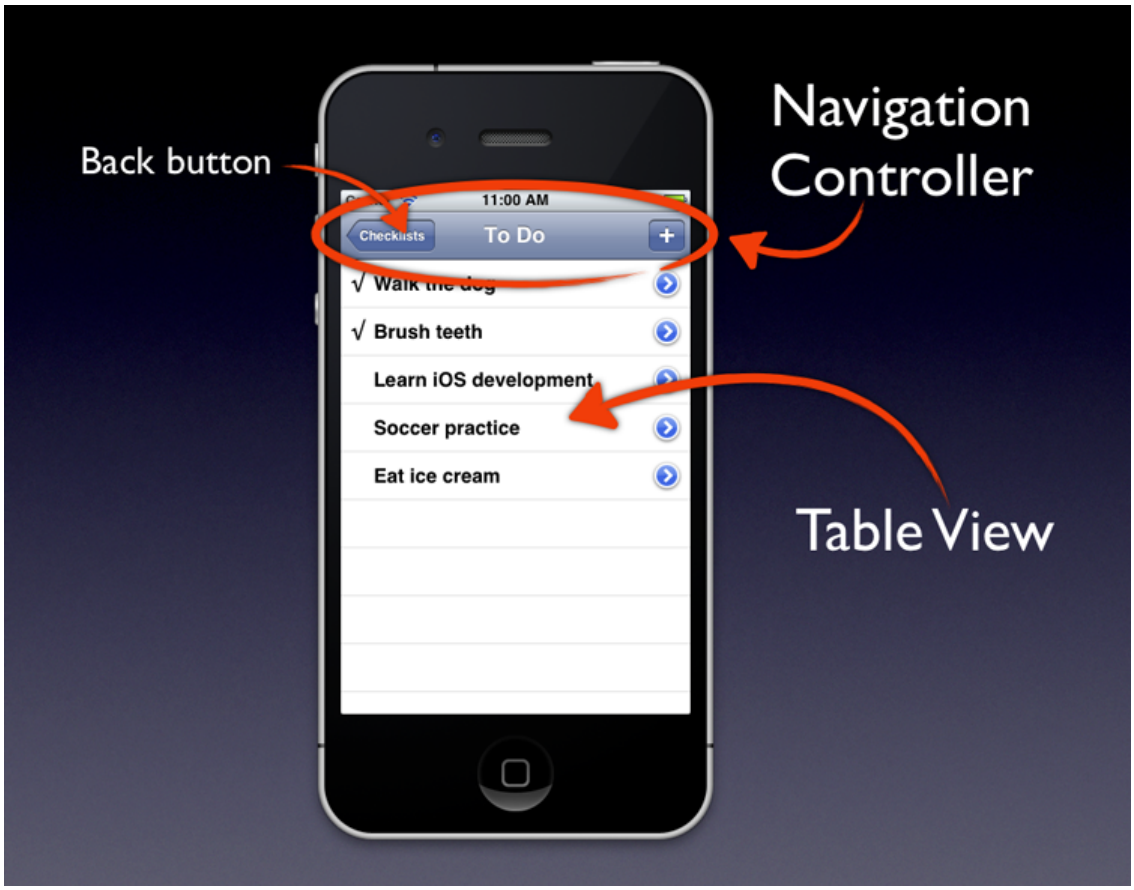
- A table view shows a list of things. All three of the screens above use a table view. In fact, all six of the app's screens are made with table views. This component is extremely versatile and the most important one to master in iOS development. With

iOS 5, table views have become easier to use and even more powerful, and we'll put that to our advantage.

- The navigation controller allows you to build a hierarchy of screens that lead from one to another. It adds a navigation bar at the top of the screen with a title and optional buttons. In our app, tapping on the name of a checklist — "To Do", for example — slides in the screen that contains the to-do items from that list. A tap on the "back button" in the upper-left corner takes you back to the previous screen with a swift animation. That is the navigation controller at work; you have no doubt seen it before in other apps.

Navigation controllers and table views are often used together:

The blue bar at the top is the navigation bar. The list is the table view.



If you take a look at the apps that come with your iPhone — Calendar, Notes, Contacts, Mail, YouTube — you'll notice that even though they look slightly different, all these apps still work in very much the same way. That's because they all use table views and navigation controllers. (Some of the apps in the picture also have a tab bar at the bottom, something we'll talk about in the next tutorial.)

These are all table views inside navigation controllers: YouTube, Calendar, Notes, iPod/Music

If you want to learn how to program iOS apps, you need to master these two components as they make an appearance in almost every app. That's exactly what we'll focus on in this tutorial. You'll also learn how to pass data from one screen to another, a very important topic that often puzzles beginners.

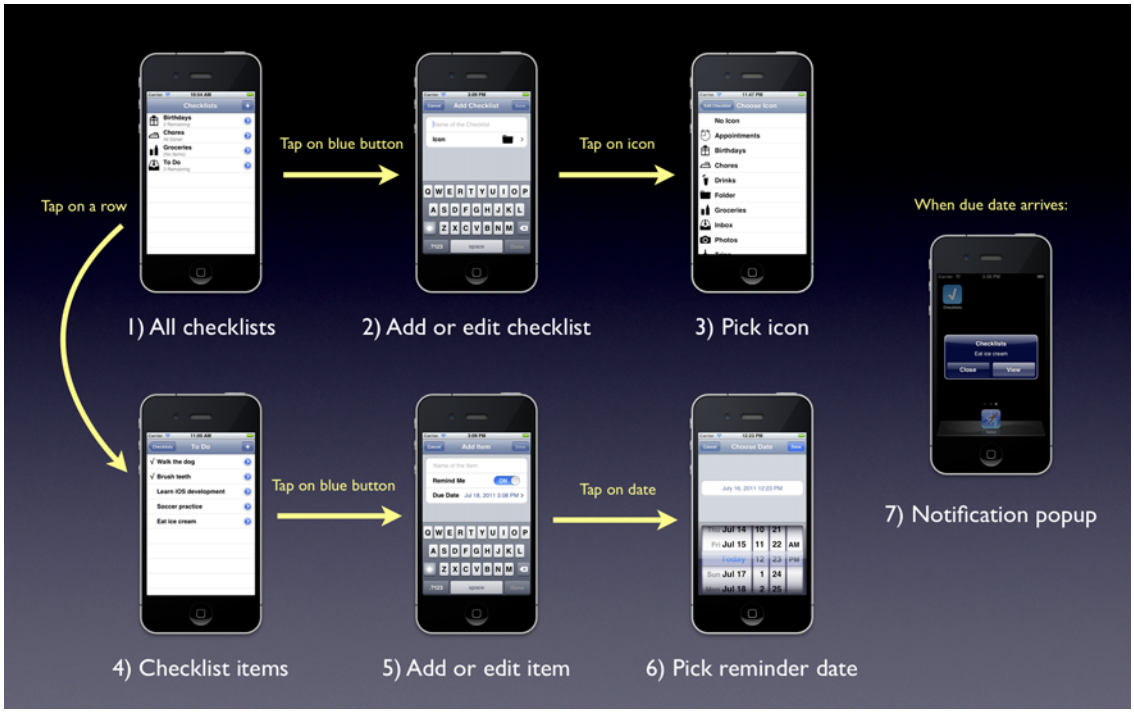
When you're done with this lesson, the concepts *view controller*, *table view* and *delegate* will be so familiar to you that you can program them in your sleep (although I hope you'll dream of other things).

This is a very long read with a lot of source code, so take your time to let it all sink in. I encourage you to experiment with the code that we will be writing. Change stuff and see what it does, even if it breaks the app. Playing with the code is the quickest way to learn!

The Checklists design

Just so you know what you're in for, here is an overview of how the Checklists app will work:

All the screens of the Checklists app



The main screen of the app shows all your checklists (1). You can create multiple lists to organize your to-do items. A checklist has a name, an icon, and zero or more items. You can edit the name and icon of a checklist in the Add/Edit Checklist screen (2) and (3).

You tap on the checklist's name to view its to-do items (4). An item has a description, a checkmark to mark the item as done, and an optional due date. You can edit the item in the Add/Edit Item screen (5) and (6).

The app uses local notifications to automatically notify the user of checklist items that have their "remind me" option set, even if the app isn't running (7). Pretty cool.

Playing with table views

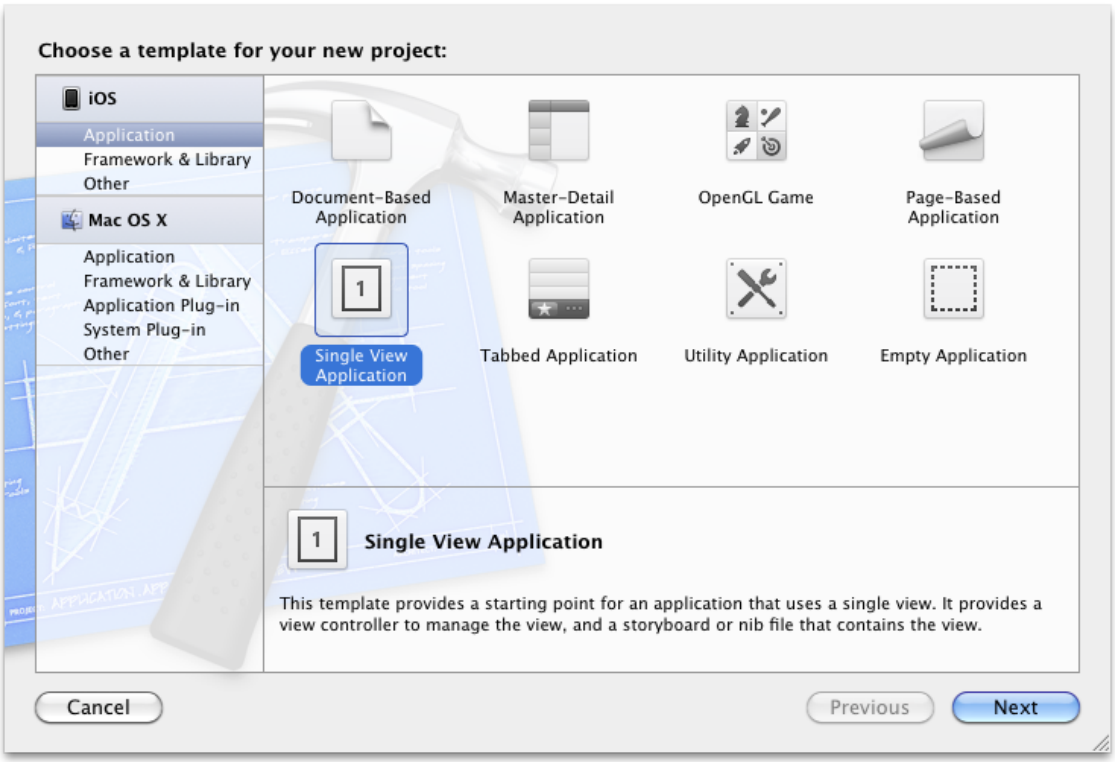
Seeing as table views are so important, we will start our app by examining how table views work. Because I always like to split up the workload into small, simple steps, this is what we're going to do in this first section:

1. Put a table view on our app's screen
2. Put data into that table view
3. Allow the user to tap on a row in the table to toggle a checkmark on and off

Once we have these basics up and running, we'll keep adding new functionality to it over the course of this tutorial until we end up with the full-blown app.

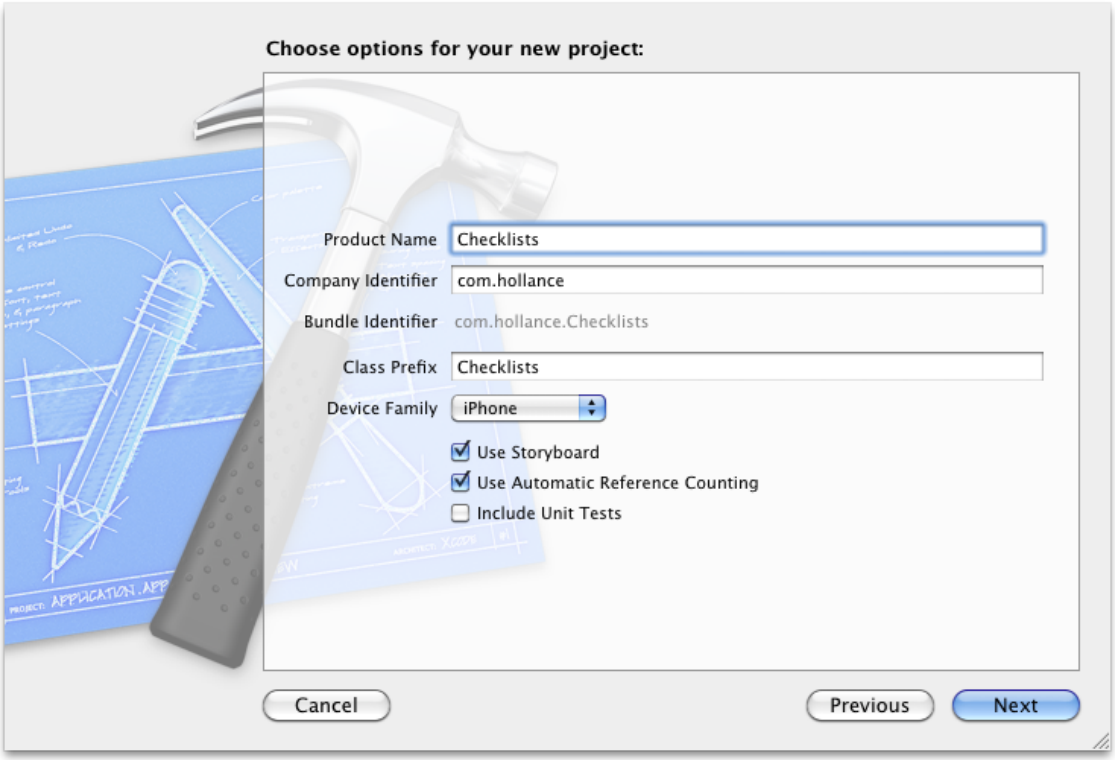
» Launch Xcode and start a new project. Choose the Single View Application template:

Choosing the Xcode template



Xcode will ask you to fill out a few options:

Choosing the template options



» Fill out these options as follows:

- Product Name: Checklists
- Company Identifier: Use your own identifier here, using reverse domain name notation
- Class Prefix: Checklists
- Device Family: iPhone
- Use Storyboard: Check this item
- Use Automatic Reference Counting: Check this item
- Include Unit Tests: Do not check this item

» Press Next and choose a location for the project.

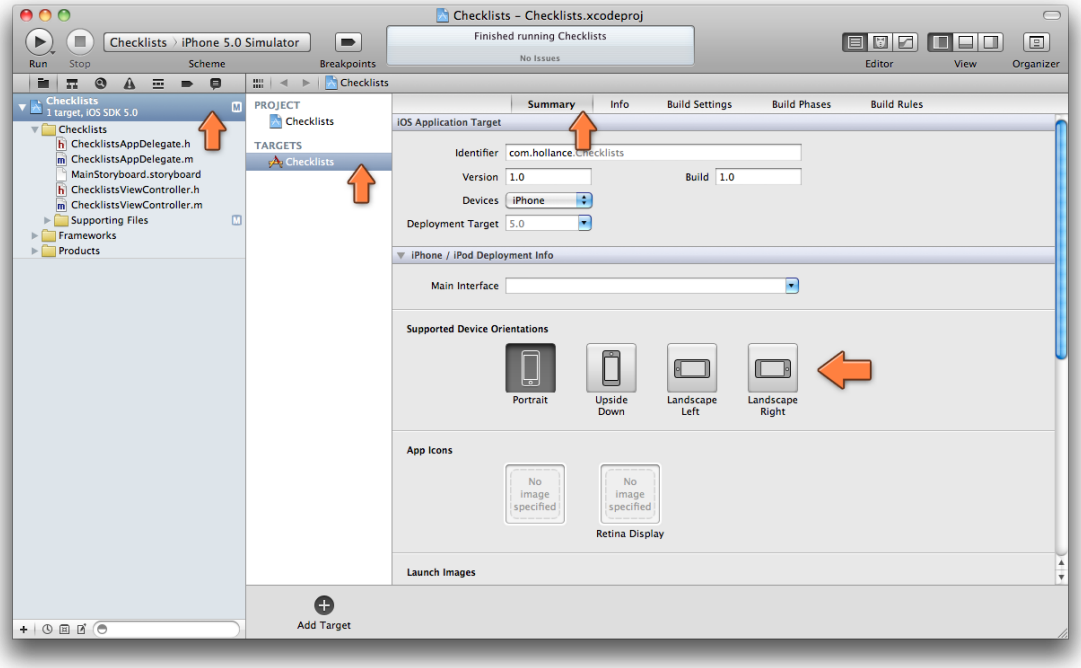
These steps are similar to what we did in the previous tutorial, except that now we enabled the Use Storyboard option.

You can run the app if you want but at this point it just consists of a white screen.

Our app will run in portrait orientation only but the project that Xcode just generated also includes the landscape orientation.

» Click on the Checklists project item at the top of the Project Navigator, then on the Checklists target and select the Summary tab. Under Supported Device Orientations, deselect the Landscape Left and Landscape Right buttons so that only Portrait is selected.

The Supported Device Orientations setting



Just changing this setting is not enough. As you've seen in the previous tutorial we also need to alter the `shouldAutorotateToInterfaceOrientation` method.

» Open ChecklistsViewController.m in the editor and make the following change:

ChecklistsViewController.m

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)↵
    interfaceOrientation
{
  return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

We've done this before, so this should be no problem for you. The difference is that this time we only allow portrait instead of landscape.

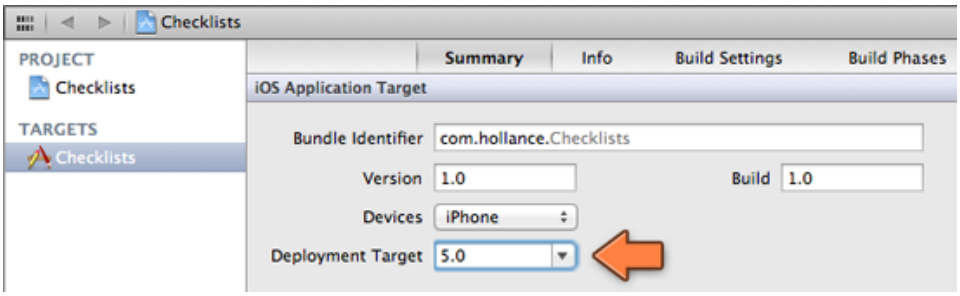
Note: Make sure it says `==` and not `!=`. The Xcode template already put in a version of this method that uses the `!=` operator, but that is the wrong way round.

Upside down

There is also a Portrait Upside Down orientation but we won't use it. If your app supports Portrait Upside Down, then users are able to rotate their iPhone so that the Home button is at the top of the screen instead of at the bottom. That may be confusing, especially when the user receives a phone call. If they were to answer with the phone upside down, the microphone is at the wrong end. iPad apps, on the other hand, are supposed to support all four orientations including portrait upside-down.

Important: If you're using Xcode 4.5 or better, make sure you set the Deployment Target to 5.0 and not 6.0 or higher:

Setting the Deployment Target version number for the project



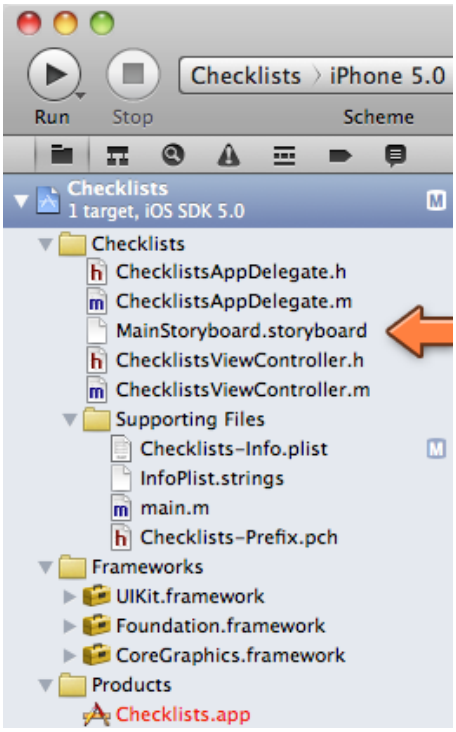
It's always best to compile against the latest version of the SDK (Software Development Kit). In Xcode 4.5 that is the iOS 6.0 SDK. You can still make apps that run on older versions of iOS, but in order to enable that you must set the Deployment Target field to the oldest version that you wish to support.

Storyboards

For this app we will use *storyboarding*, a new technique introduced with iOS 5 that makes apps easier and quicker to write. Before iOS 5 you had to make a separate nib file for each of your app's screens but with storyboarding the designs for all your view controllers are combined into a single Storyboard file.

As you can see in the Project Navigator on the left side of the screen, Xcode automatically made the ChecklistsViewController.h and .m files for us, which contain the view controller for the main screen, but there is no corresponding ChecklistsViewController.xib. Because we chose "Use Storyboard" when we created the project, this nib has been replaced by MainStoryboard.storyboard.

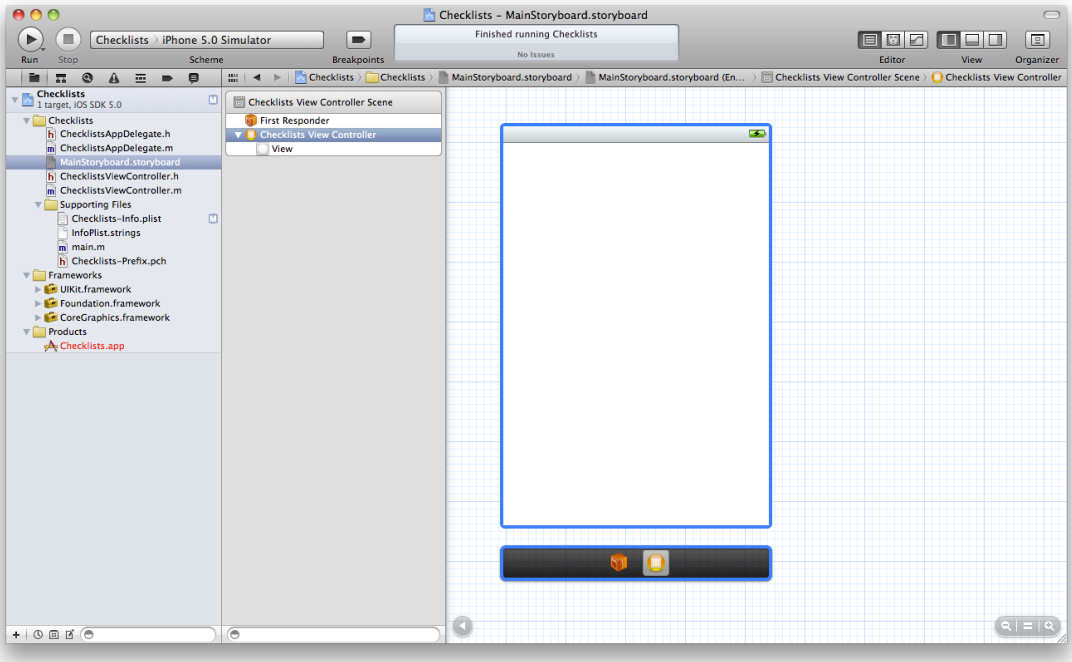
The Storyboard file in the Project Navigator



» Click on MainStoryboard.storyboard to select it.

The Xcode window turns into something that looks a lot like the Interface Builder we've seen before. This is the Storyboard editor. It works almost the same way except that you can have multiple view controllers in the same canvas. Recall that a view controller represents one screen of your app. In storyboard terminology, each view controller is named a "scene".

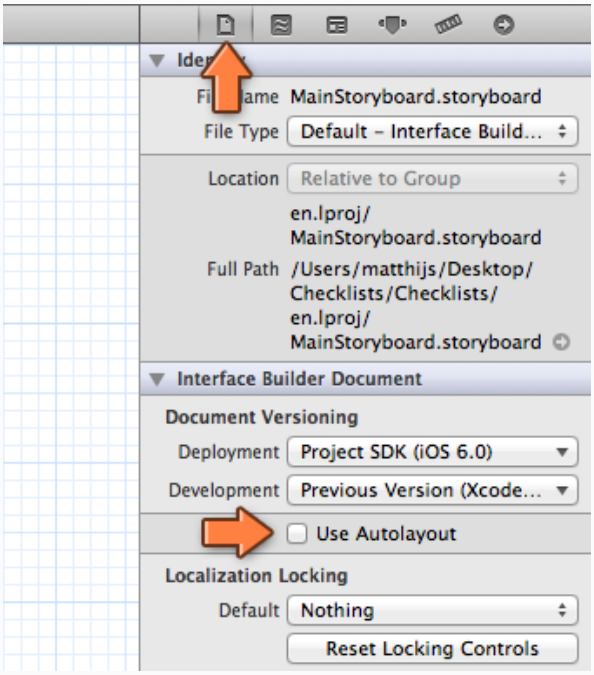
The Storyboard editor with our app's only scene



Important: If you're using Xcode 4.5 or better, you should first disable Auto Layout. This is a new feature from iOS 6 for making advanced user interface designs. Auto Layout is enabled by default on Xcode 4.5 and up, but we won't need it in these tutorials. Auto Layout only works on iOS 6 and later, so by turning it off our apps will also be able to run on iOS 5.

» Open the Utilities pane and go to the File Inspector. Uncheck the Use Autolayout option:

Disabling Auto Layout (Xcode 4.5 and better only)



The Auto Layout setting applies to the entire storyboard, so you only have to do this once.

» Remove the Checklists View Controller scene from the storyboard so that the canvas is empty (the pane to the left should say “No Scenes”).

We’re deleting this scene because we don’t want a regular view controller but a so-called *table view controller*. This is a special type of view controller that makes working with table views a little easier.

To change `ChecklistsViewController`’s type to a table view controller, we first have to edit its .h file.

» Click on ChecklistsViewController.h to open it in the source code editor and change the `@interface` line from this:

ChecklistsViewController.h

```
@interface ChecklistsViewController : UIViewController
```

to this:

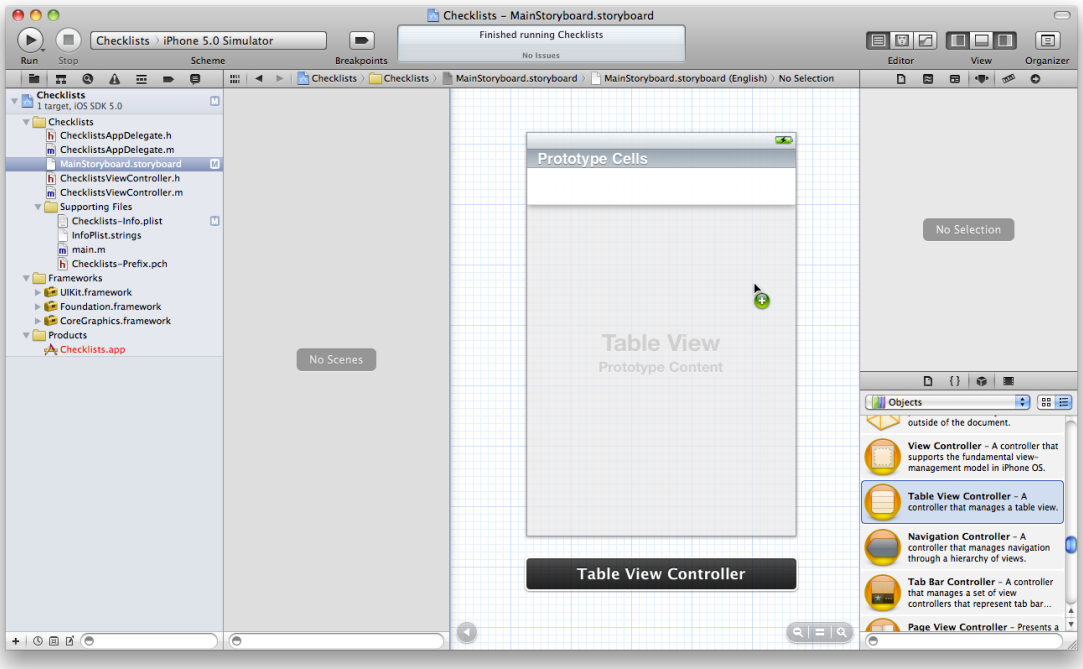
ChecklistsViewController.h

```
@interface ChecklistsViewController : UITableViewController
```

With this change we tell the Objective-C compiler that the view controller is now a `UITableViewController` instead of a regular `UIViewController`.

» Go back to the Storyboard editor and drag a Table View Controller from the Object Library (bottom-right corner) into the canvas:

Dragging a Table View Controller into the Storyboard editor

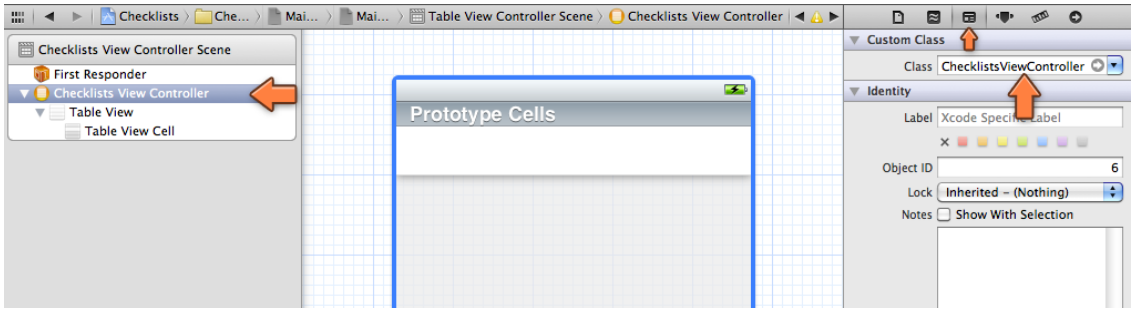


This adds a Table View Controller scene to the storyboard.

» Go to the Identity Inspector (the third tab in the inspectors pane on the right of the Xcode window) and under Custom Class type “ChecklistsViewController” (or choose it using the blue arrow).

The name of the scene in the Scene List on the left changes to “Checklists View Controller Scene”. (When you do this, make sure the actual Table View Controller is selected, not the Table View inside it. There should be a fat blue border around the scene.)

Changing the Custom Class of the Table View Controller

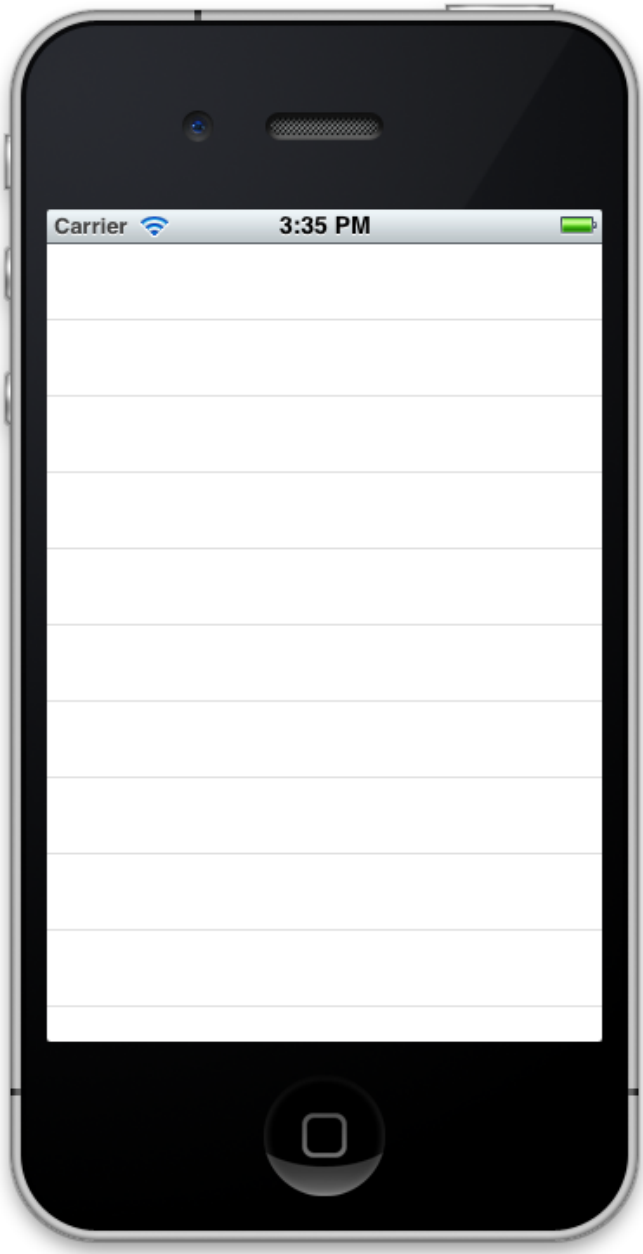


We have now changed our `ChecklistsViewController` from a regular controller into a table view controller. As its name implies, and as you can see in the Storyboard editor, the controller contains a Table View object. We'll go into the difference between controllers and views soon, but for now remember that the controller is the whole screen while the table view is the object that actually draws the list.

» Run the app.

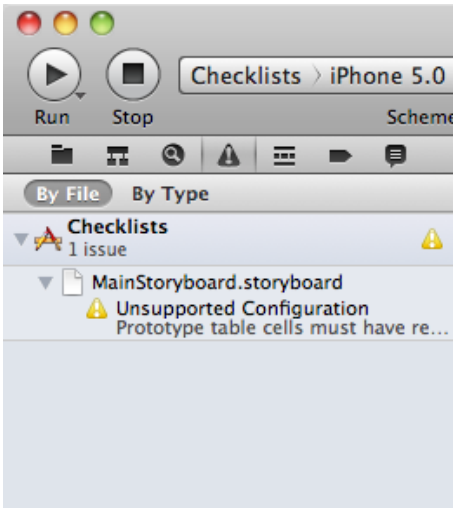
Instead of a plain white screen you'll now see an empty list. This is the table view. You can drag the list up and down but it doesn't contain any data yet.

The app now uses a table view controller



At this point Xcode may give a warning (“Unsupported Configuration: Prototype table cells must have reuse identifiers”) and we’ll fix that in a minute.

Xcode gives a warning about prototype table cells

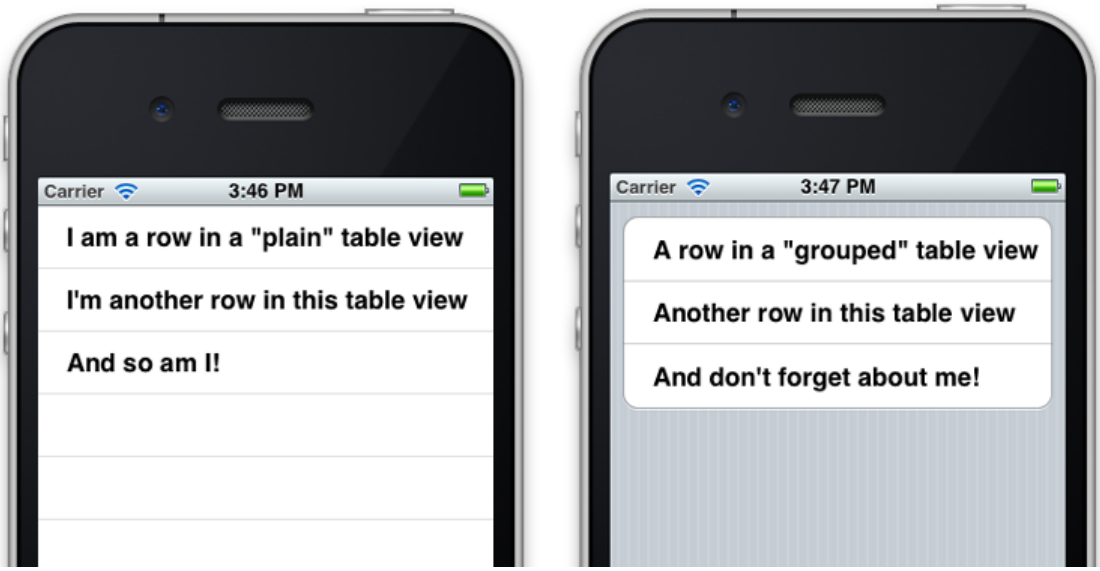


The anatomy of a table view

First, let's talk a bit more about table views. A `UITableView` object displays a list of things. I'm not sure why it's named a table because a table is commonly thought of as a spreadsheet-type thing that has multiple rows and multiple columns, whereas the `UITableView` only has rows. It's more of a list than a table, but I guess we're stuck with the name now.

There are two styles of tables: "plain" and "grouped". They work mostly the same but there are a few small differences. The most visible dissimilarity is that rows in the grouped style table are slightly offset from the edges and are placed inside rounded rectangles.

A plain-style table (left) and a grouped table (right)

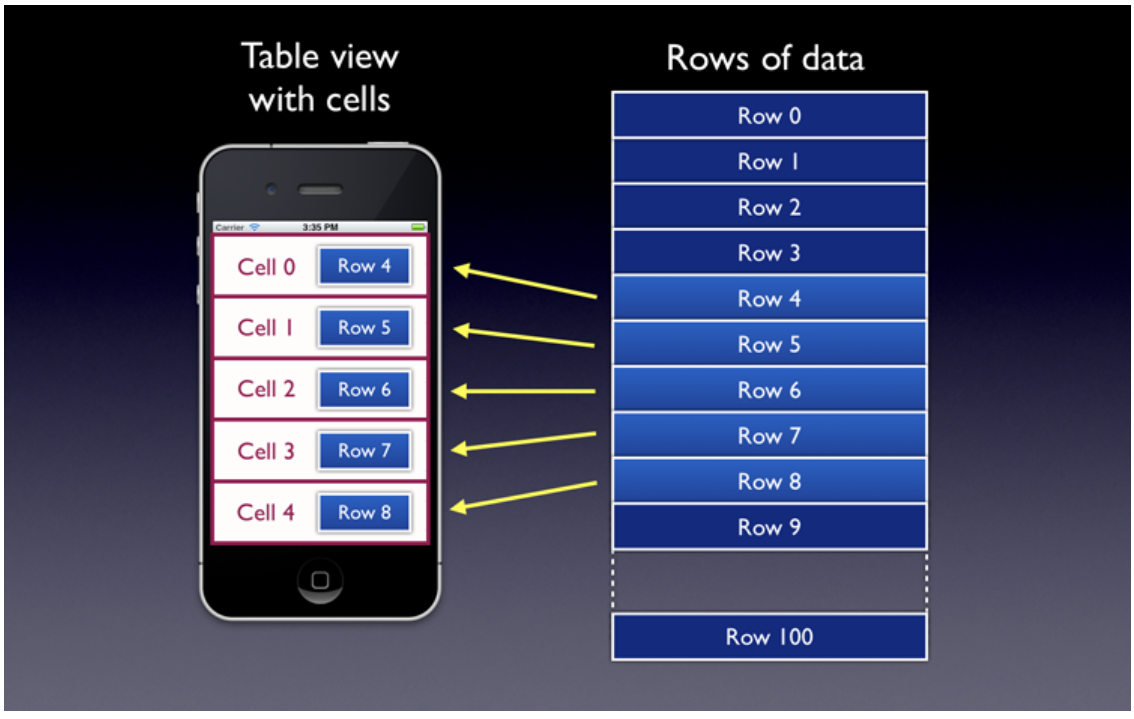


The plain style is used for rows that all represent something similar, such as contacts in an address book where each row contains the name of one person. The grouped style is used when each row represents something different, such as the various attributes of one of those contacts. The grouped style table would have a name row, an address row, a phone number row, and so on. We'll use both table styles in the Checklists app.

The data for a table comes in the form of *rows*. In the first version of our app, each row will correspond to a to-do item that you can check off when you're done with it. You can potentially have many rows (tens of thousands) although that kind of design isn't recommended. Most users will find it incredibly annoying to scroll through ten thousand rows to find the one they want, and who can blame them…

Tables display their data in *cells*. A cell is related to a row but it's not exactly the same. A cell is a view that shows a row of data that happens to be visible at that moment. If your table can show 10 rows at a time on the screen, then it only has 10 cells, even though there may be hundreds of rows with actual data. Whenever a row scrolls off the screen and becomes invisible, its cell will be re-used for a new row that scrolls into the screen.

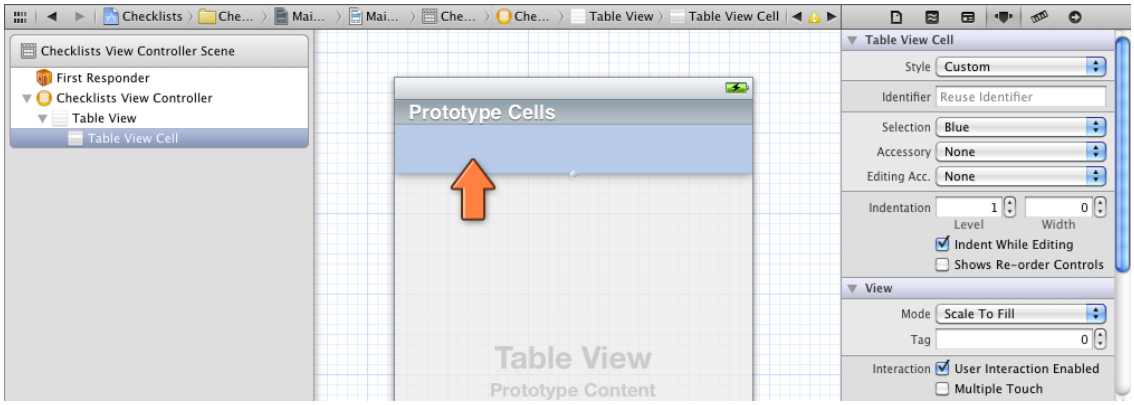
Cells display the contents of rows



Until iOS 5 you had to put in quite a bit of effort to create cells for your tables but now we have a very handy new feature named *prototype cells* that lets you design your cells visually in the Storyboard editor.

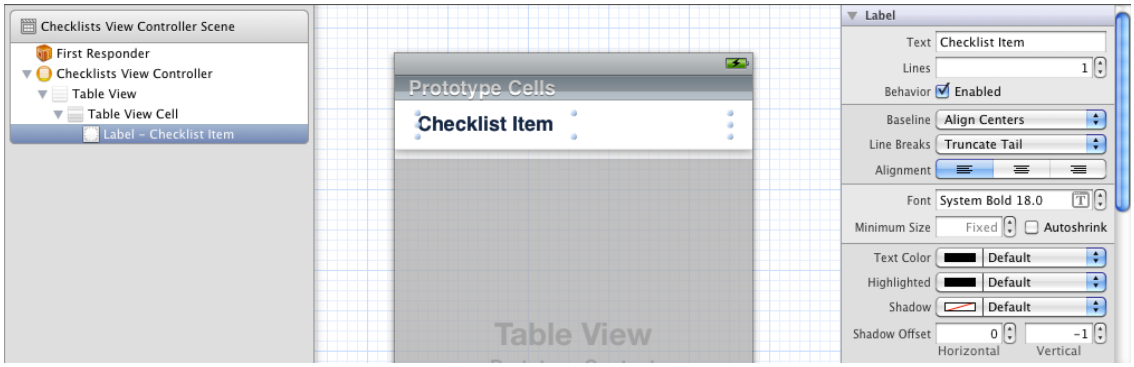
» Go to the Storyboard editor and click the empty cell to select it. It will become blue:

Selecting the prototype cell



» Drag a Label from the Object Library into this cell. Give the label some placeholder text: "Checklist Item". Set its font to System Bold, size 18. Make sure the label spans the entire width of the cell (but leave a small margin on the sides). Uncheck Autoshrink.

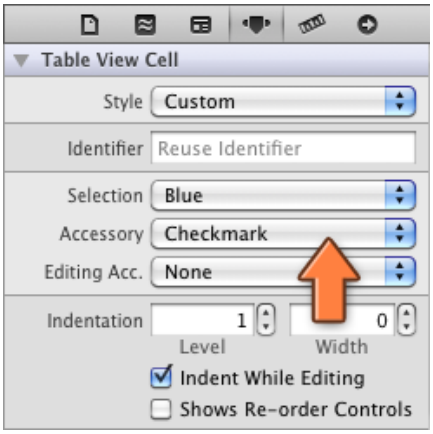
Adding the label to the prototype cell



Besides the label we will also add a checkmark to the cell's design. The checkmark is provided by something called the *accessory*, which is a built-in subview that appears on the right side of the cell. You can choose from a few standard accessory controls or provide your own.

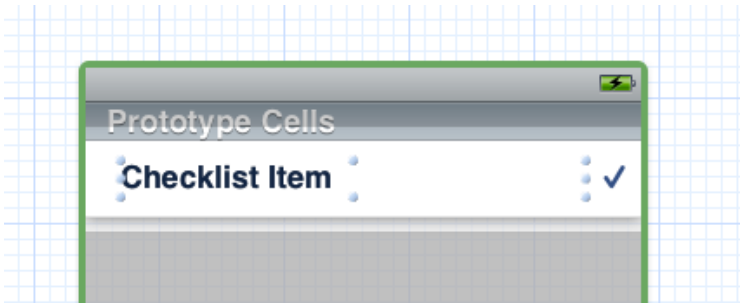
» Select the Table View Cell. Inside the Attributes Inspector set the Accessory field to Checkmark:

Changing the accessory to get a checkmark



Our design now looks like this:

The design of the prototype cell: a label and a checkmark



You may want to resize the label a bit so that it doesn't overlap the checkmark.

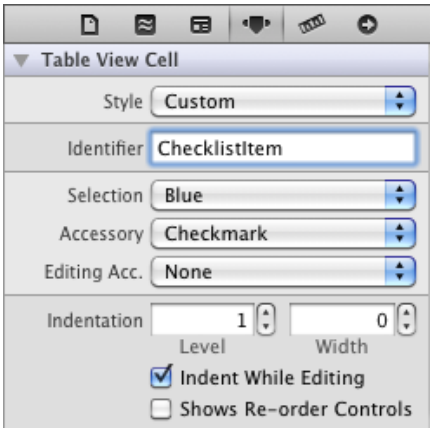
We also need to set a *reuse identifier* on the cell. This is an internal name that the table view uses to find free cells to reuse when rows scroll off the screen and new rows must become visible. The table needs to assign cells to those new rows and recycling existing cells is more efficient than creating new cells. This technique is what makes your table views scroll smoothly.

Reuse identifiers are also important for when you want to display different types of cells in the same table. For example, one type of cell could have an image and a label and another could have a label and a button. You would give each cell type its own identifier, so the table view can assign the right cell to the right row.

Our app has only one type of cell but we still need to give it an identifier.

» Type "ChecklistItem" into the Table View Cell's Identifier field. That should make Xcode shut up about the warning it gave earlier.

Giving the table view cell a reuse identifier



» Run the app and you'll see… exactly the same as before. The table is still empty.

We only added a cell design to the table, not actual rows. Remember that the cell is just the visual representation of the row, not the actual data. To add data to the table, we have to write some code.

The data source

» Head on over to ChecklistsViewController.m and add the following methods right before the `@end` line at the bottom of the file:

ChecklistsViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
{
  return 1;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"↵
      ChecklistItem"];
  return cell;
}
```

These two methods are part of `UITableView`'s *data source* protocol. The data source is the link between your data and the table view. Usually the view controller plays the role of data source and therefore implements these methods.

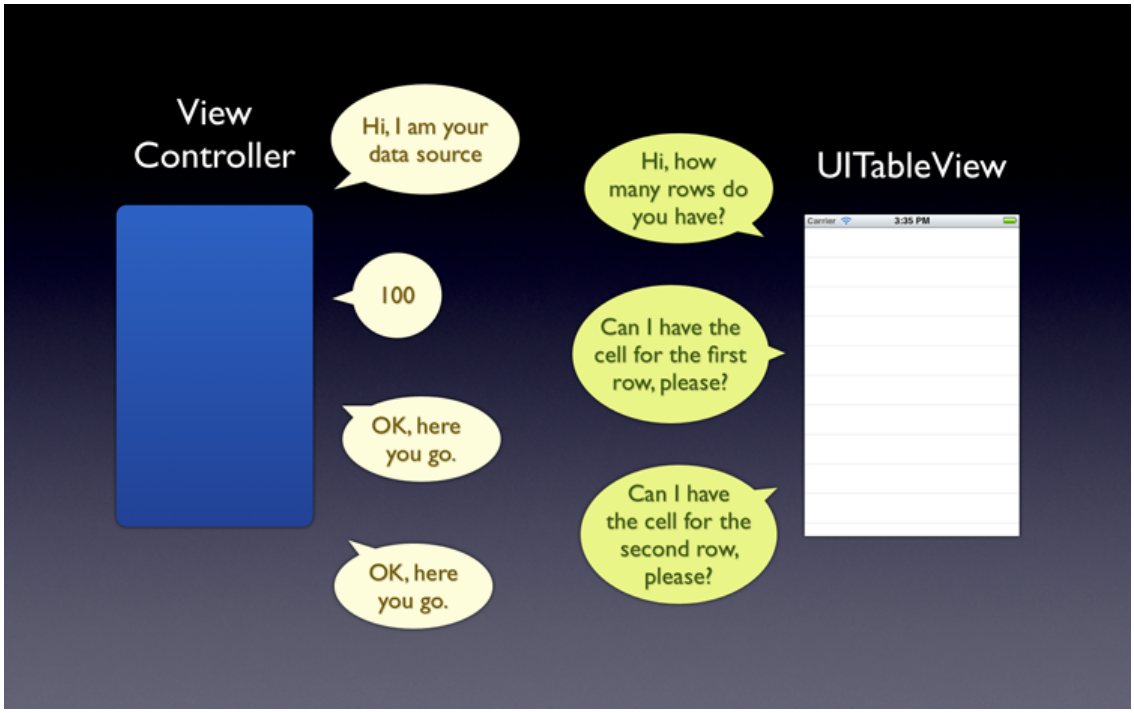
The table view needs to know how many rows of data it has and how it should display each of those rows. But you can't simply dump that data into the table view's lap and be

done with it. You don't say: "Dear table view, here are my 100 rows, now go show them on the screen." Instead, you say to the table view: "This view controller is now your data source. You can ask it questions about the data anytime you feel like it."

Once it is hooked up to a data source, the table view sends a `numberOfRowsInSection` message when it wants to know how many rows there are. And when the table view needs to display a particular row it sends the `cellForRowAtIndexPath` message to ask the data source for a cell.

You see this type of pattern all the time in iOS: one object does something on behalf of another object. In this case, the `ChecklistsViewController` works to provide the data to the table view, but only when the table view asks for it.

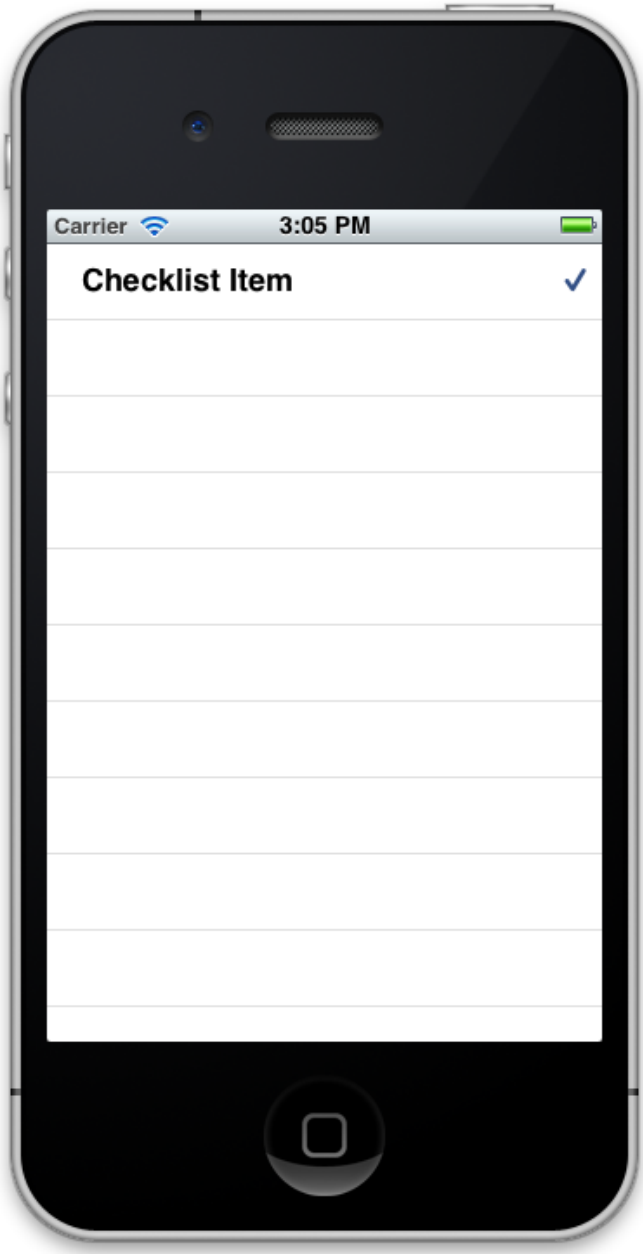
The dating ritual of a data source and a table view



Our implementation of `numberOfRowsInSection` returns the value 1. This tells the table view that we just have one row. The table view then calls the `cellForRowAtIndexPath` method to obtain a cell for that row. Inside `cellForRowAtIndexPath` we simply grab a copy of the prototype cell and give that back to the table view. This is where you would normally put the row data into the cell, but our app doesn't have any row data yet.

» If you haven't already, run the app and you'll see that we now have a row in the table:

Our table now has one row



Methods with multiple parameters

Most of the methods we have used so far took only one parameter or did not have any parameters at all, but these table view data source methods take two:

```
- (NSInteger)tableView:(UITableView *)tableView   // parameter 1
 numberOfRowsInSection:(NSInteger)section         // parameter 2
{
  . . .
}

- (UITableViewCell *)tableView:(UITableView *)tableView   // parameter 1
         cellForRowAtIndexPath:(NSIndexPath *)indexPath   // parameter 2
{
  . . .
}
```

The first parameter of both methods is the `UITableView` object in question (the table view on whose behalf these methods are invoked) and the second parameter is either the section number in the case of `numberOfRowsInSection`, or something called the "indexPath" in the case of `cellForRowAtIndexPath`.

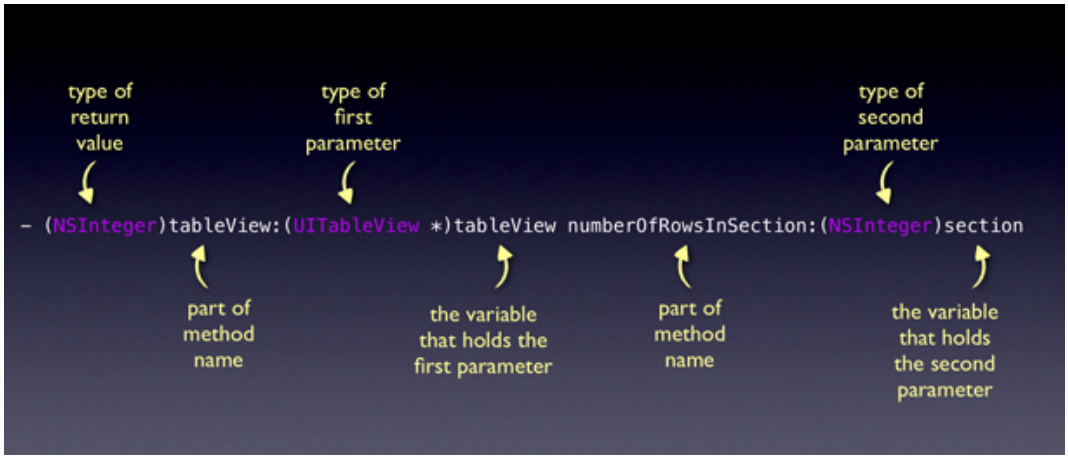
In other programming languages a method with multiple parameters typically looks like this:

```
void numberOfRowsInSection(UITableView *tableView, NSInteger section)
{
  . . .
}
```

But that's not the way we do it in Objective-C. It may look a little weird if you're coming from another language, but once you get used to it you'll find that this notation is actually quite readable.

To summarize, these are the various parts that make up a method declaration:

The different parts in a method name



The name of this method is officially `tableView:numberOfRowsInSection:` (including the colons). If you pronounce that out loud, it actually makes sense. It asks for the number of rows in a particular section in a particular table view.

Exercise: Modify the app so now it shows five rows. ■

That shouldn't have been too hard:

ChecklistsViewController.m

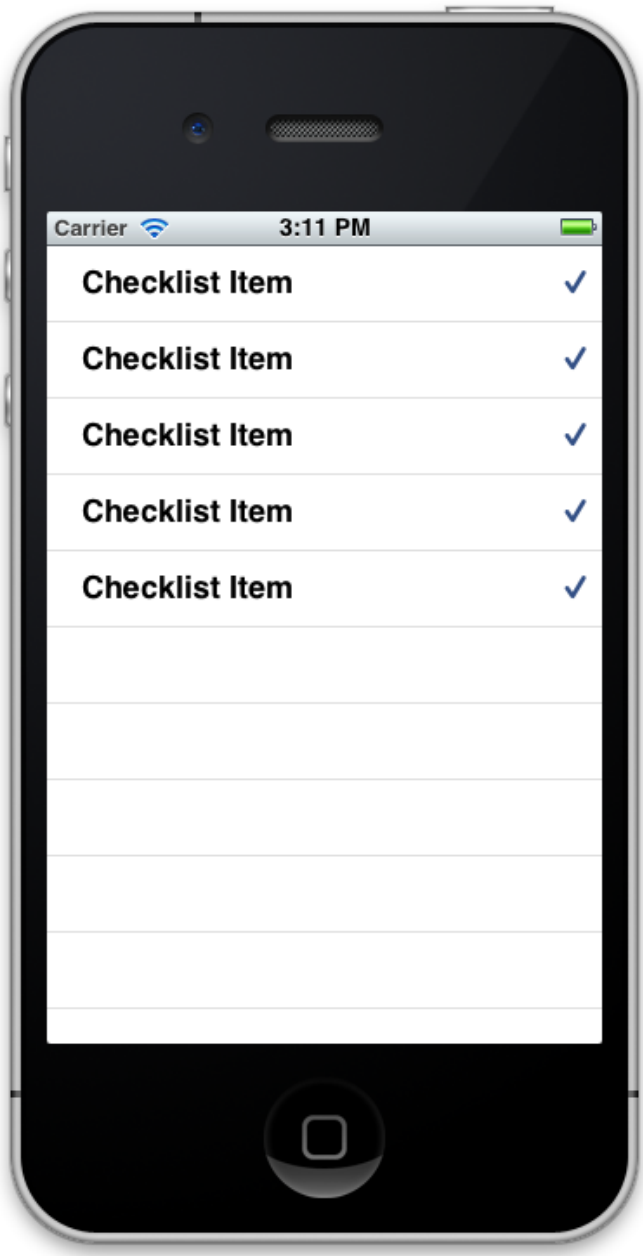
```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
{
  return 5;
}
```

If you were tempted to go into the Storyboard editor and duplicate the prototype cell there five times, then you were confusing cells with rows.

When we make `numberOfRowsInSection` return the number 5, we tell the table view that there will be five rows. The table view then calls `cellForRowAtIndexPath` five times, once for each row.

Because `cellForRowAtIndexPath` currently just returns a copy of the prototype cell, our table view now shows five identical rows:

Our table now has five identical rows



There are several ways that we can create cells in `cellForRowAtIndexPath` but by far the easiest approach is to add a prototype cell to the table view (as we've done in the Storyboard editor) and then call `[tableView dequeueReusableCellWithIdentifier]` with that cell's reuse identifier. It sounds scary but this will simply make a new copy of the prototype cell if necessary or recycle an existing cell that is no longer in use.

Once we have a cell, we should fill it up with the data from the corresponding row and give it back to the table view. That's what we'll do in the next section.

Index paths

You've seen that the table view asks the data source for a cell using the `cellForRowAtIndexPath` method. So what is an *index-path*?

`NSIndexPath` is simply an object that points to a specific row in the table. It is a combination of a row number and a section number, that's all. When the table view asks the data source for a cell, you can look at the row number inside the `indexPath.row` property to find out for which row this cell is intended.

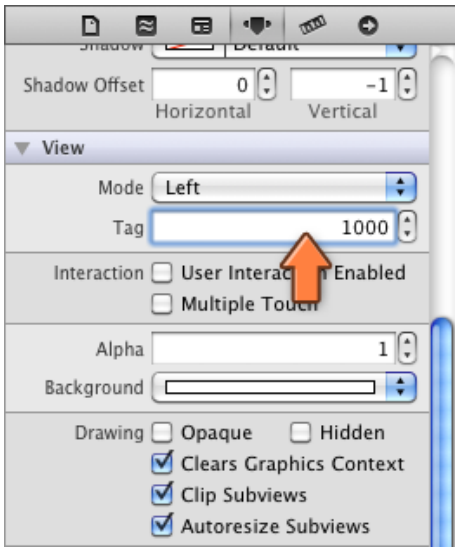
It is also possible for tables to group rows into sections. In an address book app you might sort contacts by last name. All contacts whose last name starts with "A" are grouped into their own section, all contacts whose last name starts with "B" are in another section, and so on. To find out to which section a row belongs, you would look at the `indexPath.section` property. The Checklists app has no need for this kind of grouping, so we'll ignore the `section` property of `NSIndexPath` for now.

Note: Computers start counting at 0. If you have a list of 4 items, they are counted as 0, 1, 2 and 3. It may seem a little silly at first, but that's just the way programmers do things. Therefore, for the first row in the first section, `indexPath.row` is 0 and `indexPath.section` is also 0. The second row has row number 1, the third row is row 2, and so on. Counting from 0 may take some getting used to, but after a while it becomes natural and you'll start counting at 0 even when you're out for groceries.

Currently the rows (or rather the cells) all contain the placeholder text "Checklist Item". Let's give each row a different text.

» Open the Storyboard and select the label inside the table view cell. Go to the Attributes Inspector and set the Tag field to 1000.

Set the label's tag to 1000



A *tag* is a numeric identifier that we can give to a user interface control, or any type of view really, in order to easily look it up later. Why the number 1000? No particular reason. It should be something other than 0, as that is the default value for all tags. 1000 is as good a number as any.

Note: Double check to make sure you set the tag on the *label*. It's a common mistake to set it on the table view cell itself instead of the label and then the results won't be what you expect!

» In ChecklistsViewController.m, change `cellForRowAtIndexPath` to:

ChecklistsViewController.m

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"↵
      ChecklistItem"];

  UILabel *label = (UILabel *)[cell viewWithTag:1000];

  if (indexPath.row == 0) {
    label.text = @"Walk the dog";
  } else if (indexPath.row == 1) {
    label.text = @"Brush my teeth";
  } else if (indexPath.row == 2) {
    label.text = @"Learn iOS development";
  } else if (indexPath.row == 3) {
    label.text = @"Soccer practice";
  } else if (indexPath.row == 4) {
    label.text = @"Eat ice cream";
  }

  return cell;
}
```

You've already seen the first line, which gets a copy of the prototype cell (either a new one or a recycled one) and puts it into the `cell` local variable:

```
UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"↵
    ChecklistItem"];
```

The first new line in this method is:

```
UILabel *label = (UILabel *)[cell viewWithTag:1000];
```

Here we ask the table view cell for the view with tag 1000. That is the tag we just set on our label in the Storyboard editor, so this returns a reference to that `UILabel` object. Using tags is a handy trick to get a reference to a control without having to make a `@property` for it.

Exercise: Why can't we simply add an outlet to the view controller and connect the cell's label to that property in the Storyboard editor? ■

Answer: There will likely be more than one cell in the table (at least enough of them to cover all the visible rows) and each cell will have its own label. If we connected the label from the prototype cell to an outlet on the view controller, that property would only refer to the label from one of these cells, not all of them.

Since the label belongs to the cell and not to the view controller as a whole, we can't make an outlet for it on the view controller. (That doesn't mean you cannot use properties with table view cells at all. In the MyLocations tutorial [http://www.raywenderlich.com/ios-apprentice] I'll show you how to use properties for the controls in your table view cells.)

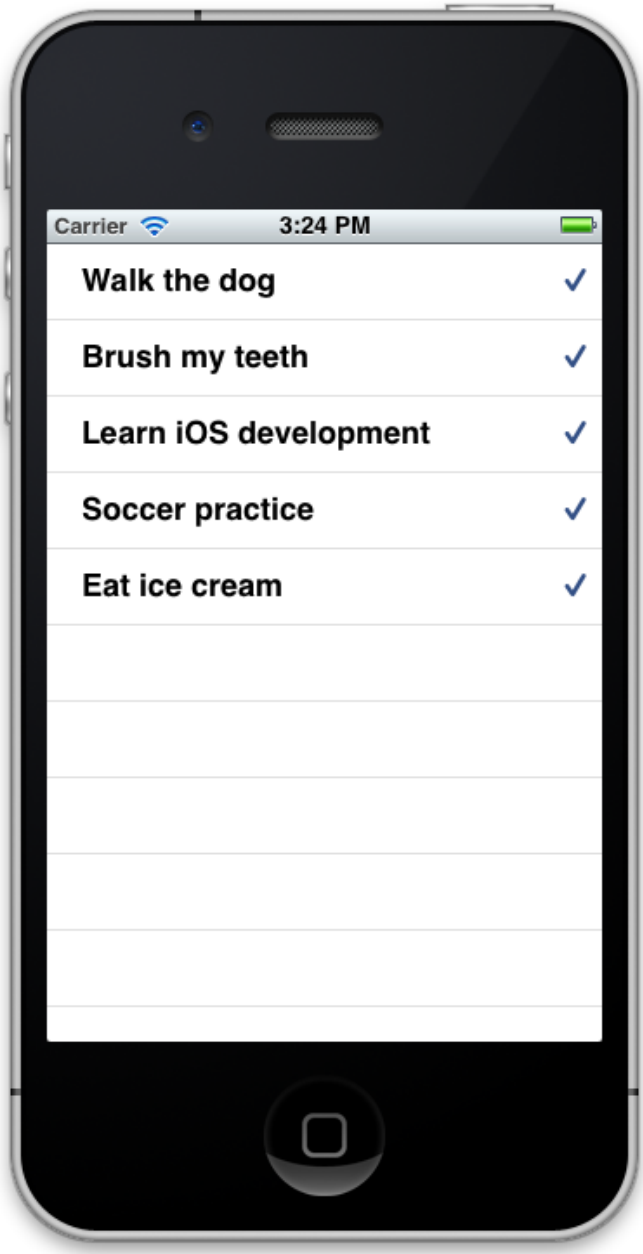
Back to the code. The next bit shouldn't give you too much trouble:

```
if (indexPath.row == 0) {
  label.text = @"Walk the dog";
} else if (indexPath.row == 1) {
  label.text = @"Brush my teeth";
} else if (indexPath.row == 2) {
  label.text = @"Learn iOS development";
} else if (indexPath.row == 3) {
  label.text = @"Soccer practice";
} else if (indexPath.row == 4) {
  label.text = @"Eat ice cream";
}
```

You have seen this `if else if else` structure before. We simply look at the value of `indexPath.row`, which contains the row number, and change the label's text accordingly. The cell for the first row — remember that we start counting at index 0 — gets the text "Walk the dog", the cell for the second row gets the text "Brush my teeth", and so on.

» Run the app and see that we have five rows, each with their own text:

The rows in the table now have their own text



That is how you write the `cellForRowAtIndexPath` method to provide data to the table. You first get a `UITableViewCell` object and then change the contents of that cell based on the row number from `NSIndexPath`.

Just for the fun of it, let's put 100 rows into the table.

» Change the code to the following:

ChecklistsViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
{
  return 100;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"↵
      ChecklistItem"];

  UILabel *label = (UILabel *)[cell viewWithTag:1000];

  if (indexPath.row % 5 == 0) {
    label.text = @"Walk the dog";
  } else if (indexPath.row % 5 == 1) {
    label.text = @"Brush my teeth";
  } else if (indexPath.row % 5 == 2) {
    label.text = @"Learn iOS development";
  } else if (indexPath.row % 5 == 3) {
    label.text = @"Soccer practice";
  } else if (indexPath.row % 5 == 4) {
    label.text = @"Eat ice cream";
  }

  return cell;
}
```

It is mostly the same as before, except that `numberOfRowsInSection` returns 100 and `cellForRowAtIndexPath` uses a slightly different method to determine which text to display where:

```
if (indexPath.row % 5 == 0) {
} else if (indexPath.row % 5 == 1) {
} else if (indexPath.row % 5 == 2) {
} else if (indexPath.row % 5 == 3) {
} else if (indexPath.row % 5 == 4) {
}
```

This uses the modulo operator `%`, which we've used in Bull's Eye to help generate random numbers, to determine what row we're on. The first row, as well as the sixth, eleventh, sixteenth and so on, will show the text "Walk the dog". The second, seventh and twelfth row will show "Brush my teeth". The third, eight and thirteenth row will show "Learn iOS Development". And so on…

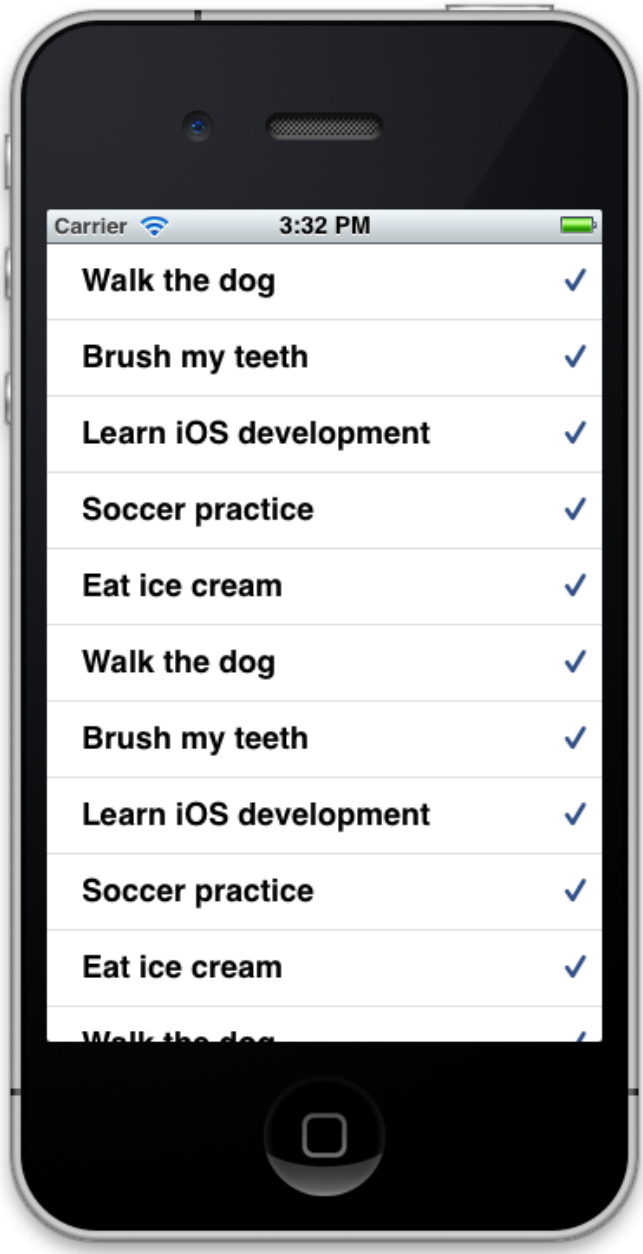
I think you get the picture, every five rows we repeat these lines. Rather than typing in all the possibilities all the way up to a hundred, we let the computer calculate this for us (that is what they are good at):

```
First row:     0 % 5 = 0
Second row:    1 % 5 = 1
Third row:     2 % 5 = 2
Fourth row:    3 % 5 = 3
Fifth row:     4 % 5 = 4
Sixth row:     5 % 5 = 0  (same as first row)  *** The sequence repeats here
Seventh row:   6 % 5 = 1  (same as second row)
Eighth row:    7 % 5 = 2  (same as third row)
Ninth row:     8 % 5 = 3  (same as fourth row)
Tenth row:     9 % 5 = 4  (same as fifth row)
Eleventh row: 10 % 5 = 0  (same as first row)  *** The sequence repeats again
Twelfth row:  11 % 5 = 1  (same as second row)
and so on...
```

If this makes no sense to you at all, then feel free to ignore it. We're just using this trick to quickly get a large table filled up.

» Run the app and you should see this:

The table now has 100 rows



Exercise: How many cells do you think this table view uses? ■

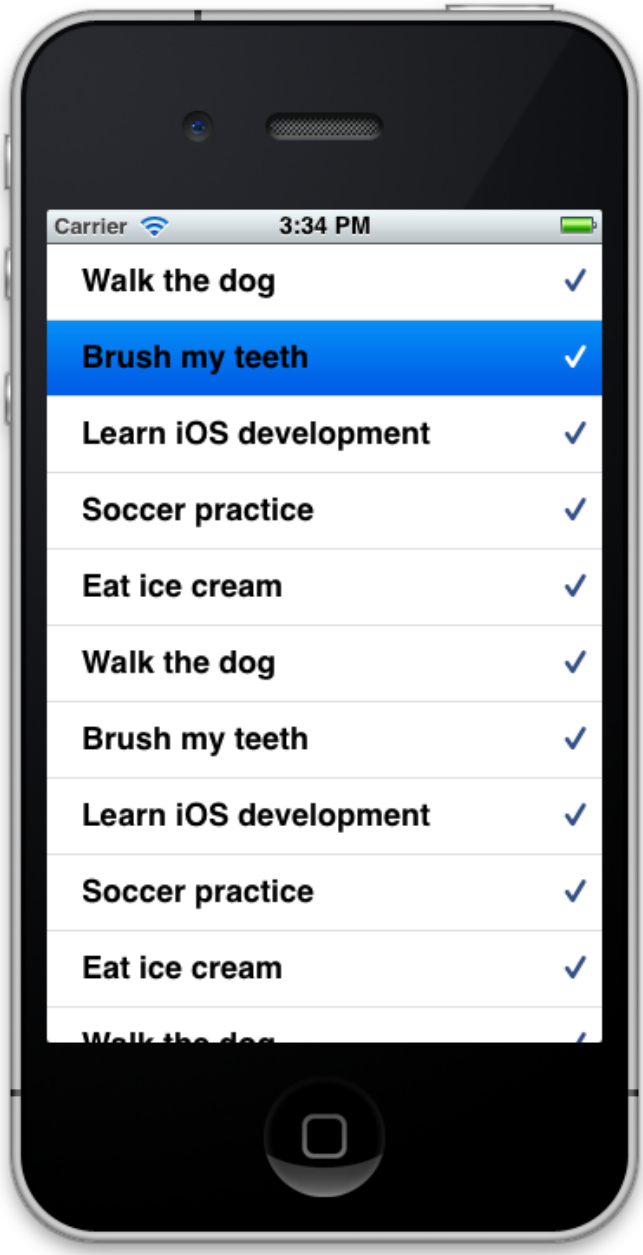
Answer: There are 100 rows but only 12 fit on the screen at a time. If you count the number of visible rows in the screenshot above you'll get up to 11 — or rather 10-and-a-half because the last cell is only partially visible — but it's possible to scroll the table in such a way that the top cell is still visible and a new cell is pulled in from below. So that makes at least 12 cells.

If you scroll really fast, then I guess it is possible that the table view needs to make a few more temporary cells, but I'm not sure about that. Is this important to know? Not really. You should let the table view take care of juggling the cells behind the scenes. All you have to do is give the table view a cell when it asks for it and fill it up with the data from the corresponding row.

Tapping on the rows

When you tap a row, notice that it colors blue. The checkmark turns white but the label text doesn't. When you let go of the row, it stays selected. We are going to change this so that tapping the row will toggle the checkmark on and off.

A tapped row stays blue



Taps on rows are handled by the table view's *delegate*. Remember that I said before that in iOS you often find objects doing something on behalf of other objects? The data source is one example of this, but the table view also depends on another little helper, the table view delegate.

The delegation pattern

The concept of delegation is very common in iOS. An object will often rely on another object to help it out with certain tasks. This *separation of concerns* keeps the system simple as each object does only what it is good at and lets other objects take care of the rest. The table view offers a great example of this.

Because every app has its own requirements for what its data looks like, the table view must be able to deal with lots of different types of data. Instead of making the table view very complex, or requiring that you modify it to suit your own apps, its designers have chosen to delegate the duty of filling up the cells to another object, the data source.

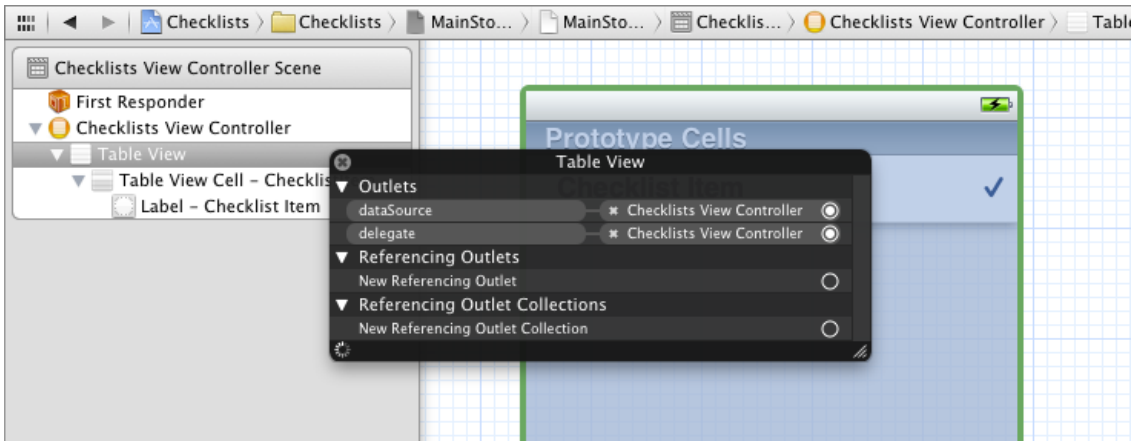
The table view doesn't really care who its data source is or what kind of data your app deals with, just that it can send the `cellForRowAtIndexPath` message and that it will receive a cell in return. This keeps the table view component simple and moves the responsibility for handling the data to where it belongs: in your code.

Likewise, the table view knows how to recognize when the user taps a row, but what it should do in response completely depends on the app. In our app we'll make this toggle the checkmark but another app will likely do something totally different. Using the delegation system, the table view can simply send a message that a tap occurred and let the delegate sort it out.

Usually components will have just one delegate but the table view splits up its delegate duties into two separate helpers: the `UITableViewDataSource` for putting rows into the table, and the `UITableViewDelegate` for handling taps on the rows and several other tasks. (Sometimes it's not entirely clear to which of these a particular piece of functionality belongs so you may have to check the documentation for both.)

If you look at the Storyboard and Ctrl-click on the table view, you can see that the table view's data source and delegate are both connected to the view controller. That is standard practice for a `UITableViewController`. (You can also use table views in regular view controllers but then you'll have to connect the data source and delegate manually.)

The table's data source and delegate are hooked up to the view controller



» Add the following method to ChecklistsViewController.m, just before `@end`:

ChecklistsViewController.m

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

When you now run the app and tap a row, you'll see that the row briefly turns blue and then becomes de-selected again.

» Let's make `didSelectRowAtIndexPath` toggle the checkmark, so change it to:

ChecklistsViewController.m

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
  UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];

  if (cell.accessoryType == UITableViewCellAccessoryNone) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

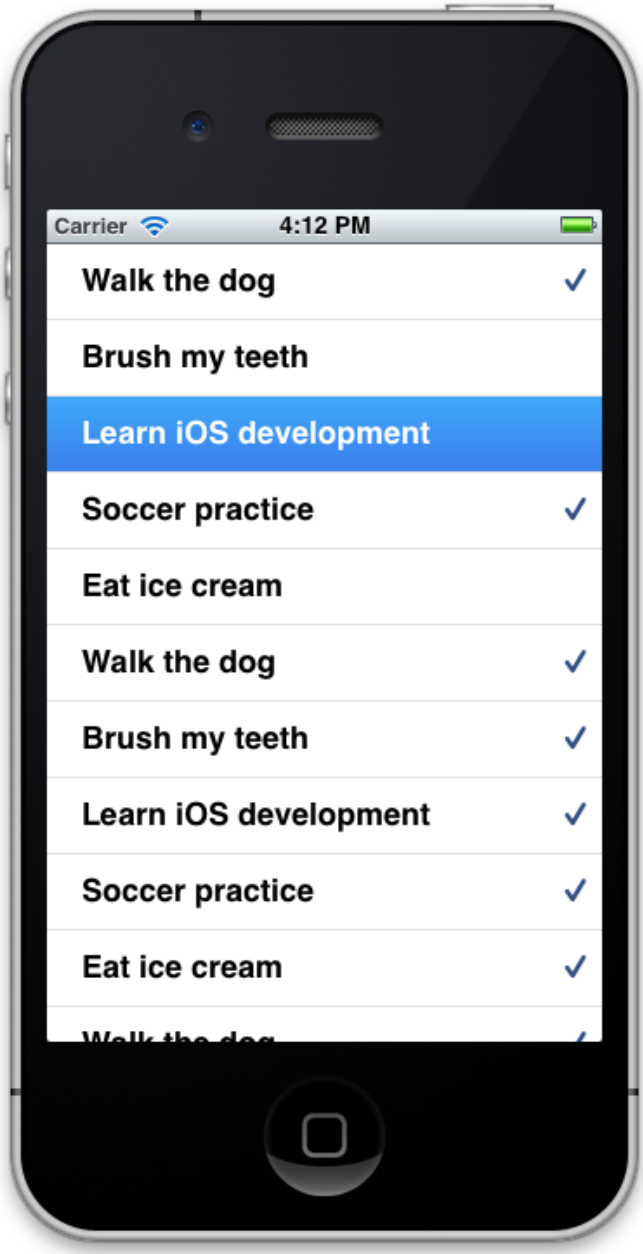
First, we get the `UITableViewCell` object in question. We simply ask the table view: what is the cell at this `indexPath` you've given me? (Note that we call `cellForRowAtIndexPath` on the table view, not on our view controller. They are different methods with the same name in different objects.)

Then we look at the cell's accessory, which you can find with the `accessoryType` property. If it is "none", then we change the accessory to a checkmark; if it was a checkmark, we change it back to none.

One more thing, go to the Storyboard editor and select the label from the table view cell. Set its Highlighted color to white. This is the color that the label text is drawn in when you tap on the row. By default it is black but white looks better on the dark blue background. (If you're using Xcode 4.3, the Highlighted color is already white, so that saves you some work.)

» Run the app and try it out. You should be able to toggle the checkmarks on the rows.

You can now tap on a row to toggle the checkmark



Sweet. However, you may have noticed there is a problem with our app. Here's how to reproduce it:

» Tap a row to remove the checkmark. Scroll that row off the screen and now scroll back (try scrolling really fast). The checkmark has reappeared! In addition, the checkmark seems to spontaneously disappear from other rows. What is going on here?

Again it's the story of cells vs. rows: we have toggled the checkmark on the cell but the cell may be reused for another row when we're scrolling. Whether a checkmark is set or not should be a property of the row, not the cell. Instead of using the cell's accessory to remember whether we should show a checkmark or not, we need some way to keep track of the checked status for each row. That means it's time to expand our data source and make it use a proper data model.

Phew! That was a lot of new stuff to take in, so I hope you're still with me. If not, then take a break and start at the beginning again. You're being introduced to a whole bunch of new concepts all at once and that can be overwhelming. But don't fear, it's OK if not everything makes perfect sense yet. As long as you get the gist of what's going on, you're good to continue. If you want to check your work, you can find the project files for the app up to this point under "01 - Table View" in the tutorial's Source Code folder.

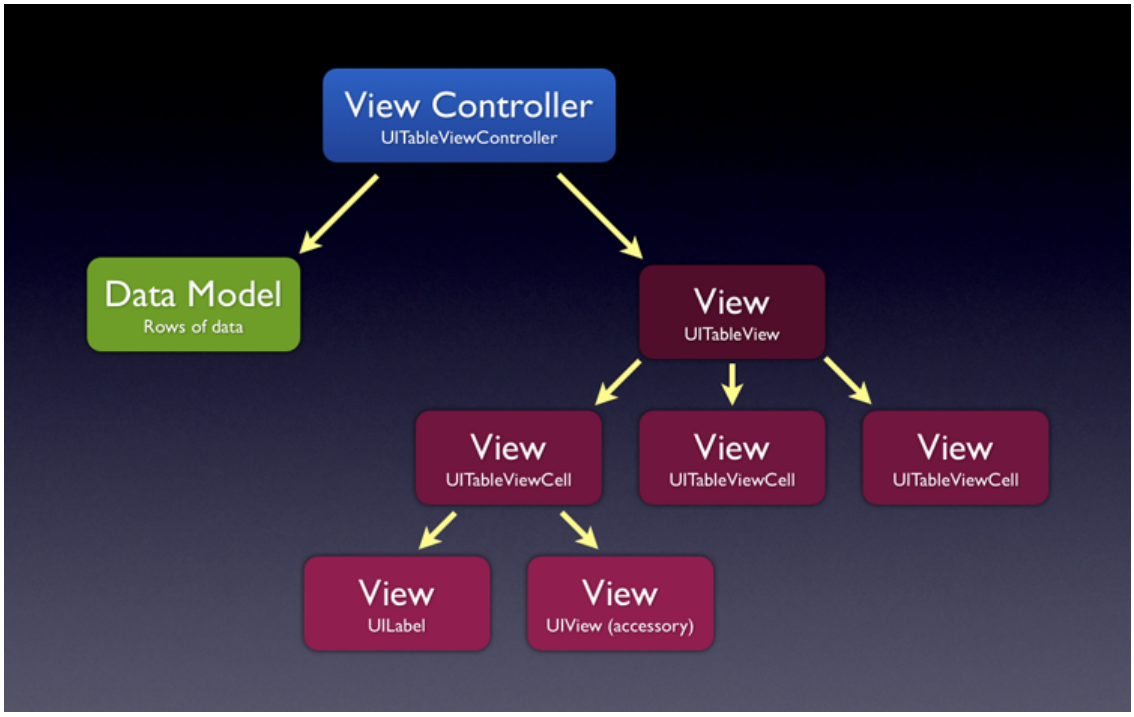
Model-View-Controller

No tutorial on programming for iOS can escape an explanation of Model-View-Controller, or MVC for short. MVC is one of the three fundamental design patterns of iOS. You've already seen the other two: delegation, making one object do something on behalf of another; and target-action, connecting events such as button taps to action methods.

Model-View-Controller roughly means that all objects in your app can be split up into three groups:

- **Model objects.** These objects contain your data and any operations on the data. For example, if you are writing a cookbook app, the model would consist of the recipes. In a game it would be the design of the levels, the score of the player and the positions of the monsters. The operations that the data model objects perform are sometimes called the *business rules* or the *domain logic*. In our app, the checklists and their to-do items form the data model.
- **View objects.** These objects make up the visual part of the app: images, buttons, labels, text fields, and so on. In a game the views are the visual representation of the game world, such as the monster animations and a frag counter. A view can draw itself and responds to user input, but it typically does not handle any application logic. Many views, such as `UITableView`, can be re-used in many different apps because they are not tied to a specific data model.
- **Controller objects.** The view controller is the object that connects your data model objects to the views. It listens to taps on the views, makes the data model objects do some calculations in response, and updates the views to reflect the new state of your model. The view controller is in charge.

Conceptually, this is how these three building blocks fit together:



The view controller has one main view, accessible through its `self.view` property, that contains a bunch of subviews. It is not uncommon for a screen to have dozens of views all at once. The top-level view usually fills the whole screen. You design the layout of the view controller's screen in a nib file or a Storyboard.

In our app, the main view is the `UITableView` and its subviews are the table view cells. Each cell also has several subviews of its own, namely the text label and the accessory.

A view controller handles one screen of the app. If your app has more than one screen, each of these has its own views and is handled by its own view controller. Your app flows from one view controller to the other.

You will often need to create your own view controllers but iOS also comes with ready-to-use view controllers, such as the mail compose controller that lets you write email, the image picker controller for photos, and the tweet sheet for sending Twitter messages.

Views vs view controllers

Note that a view and a view controller are two different things. A view is an object that draws something on the screen, such as a button or a label. The view is what you see, the view controller is what does the work behind the scenes.

I see a lot of beginners give their view controllers names such as `FirstView` and `SecondView`. Don't do that, it is very confusing! If something is a view controller, call it "ViewController" and not "View".

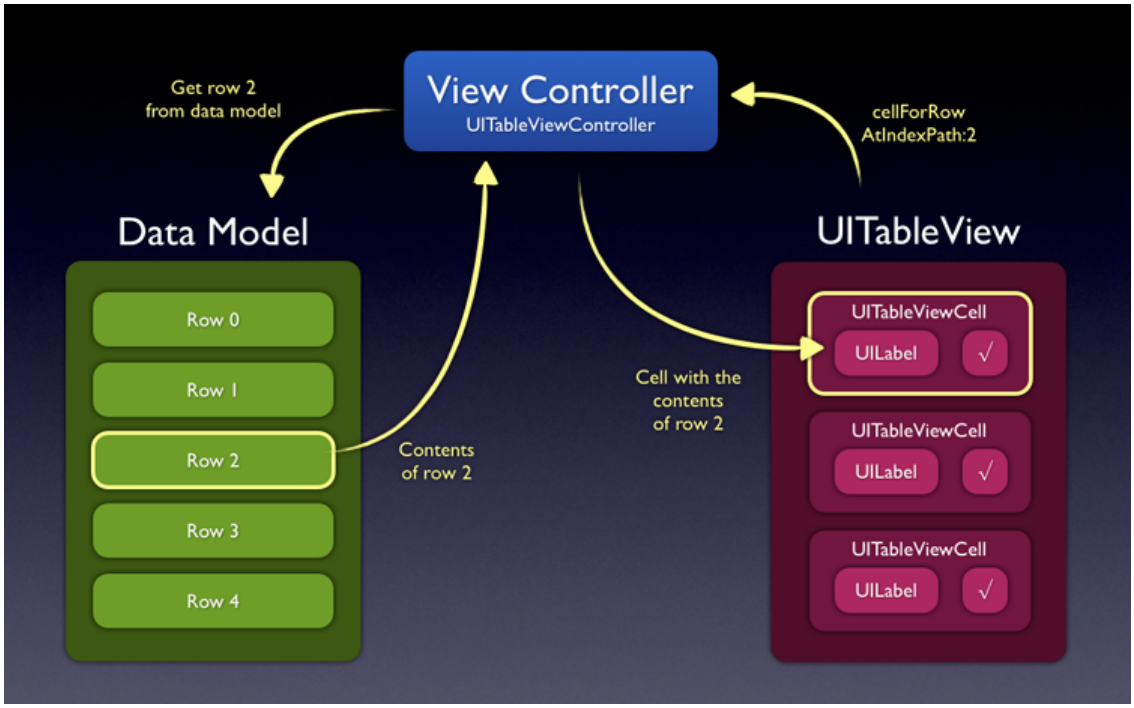
I sometimes wish Apple had left the word "view" out of "view controller" and just called it "controller" as that is a lot less misleading. The view controller doesn't just control a view, it also controls your model. It is the bridge that sits between the two.

Creating our data model

So far we've put a bunch of fake data into the table view. As you saw, we cannot just use the cells to remember our data as cells get re-used all the time and their old contents get overwritten. The cell is part of the view and is just used to display the data, but that data actually comes from somewhere else: the data model.

The rows are the data, the cells are the views. The table view controller is the thing that ties them together as it implements the table view's data source and delegate methods.

The table view controller (data source) gets the data from the model and puts it into the cells



The data model for our app consists of a list of to-do items. Each of these items will get its own row in the table. For each to-do item we need to store two pieces of information: the text ("Walk the dog", "Brush my teeth", "Eat ice cream") and whether the checkmark is set or not. That is two pieces of information per row, so we need two variables for each row.

First I'll show you the cumbersome way to program this. Note: this is what you shouldn't do. It will work but it isn't very smart. Even though this is the wrong approach, I'd still like you to follow along and copy-paste the code into Xcode and run the app. You need to understand why this approach is bad so you'll be able to appreciate the proper solution better.

» In ChecklistsViewController.m, add the following instance variables after the `@implementation` line:

ChecklistsViewController.m

```
@implementation ChecklistsViewController {
  NSString *row0text;
  NSString *row1text;
  NSString *row2text;
  NSString *row3text;
  NSString *row4text;
}
```

» Change the `viewDidLoad` method into the following:

ChecklistsViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  row0text = @"Walk the dog";
  row1text = @"Brush teeth";
  row2text = @"Learn iOS development";
  row3text = @"Soccer practice";
  row4text = @"Eat ice cream";
}
```

» Change the data source methods into:

ChecklistsViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
```

```
{
  return 5;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"↵
      ChecklistItem"];

  UILabel *label = (UILabel *)[cell viewWithTag:1000];

  if (indexPath.row == 0) {
    label.text = row0text;
  } else if (indexPath.row == 1) {
    label.text = row1text;
  } else if (indexPath.row == 2) {
    label.text = row2text;
  } else if (indexPath.row == 3) {
    label.text = row3text;
  } else if (indexPath.row == 4) {
    label.text = row4text;
  }

  return cell;
}
```

» Run the app. It still shows the same five rows as before.

What have we done here? For every row we have added an instance variable with the text for that row. Those five instance variables are our data model.

In `cellForRowAtIndexPath` we look at `indexPath.row` to figure out which row we're supposed to draw, and put the text from the corresponding instance variable into the cell.

Let's fix the checkmark toggling logic. We no longer want to toggle the checkmark on the cell but on the row. To do this, we add five new instance variables to keep track of the "checked" state of each of our rows.

» Add the following instance variables:

ChecklistsViewController.m

```
@implementation ChecklistsViewController {
  NSString *row0text;
  NSString *row1text;
  NSString *row2text;
  NSString *row3text;
```

```
  NSString *row4text;

  BOOL row0checked;
  BOOL row1checked;
  BOOL row2checked;
  BOOL row3checked;
  BOOL row4checked;
}
```

You may have seen the `BOOL` symbol a few times before, but this is the first time we're using it for variables. `BOOL` is a datatype just like `int` and `NSString`, except that it can hold only two possible values: `YES` and `NO`. In other languages these are commonly called "true" and "false" but Objective-C uses the simpler terms `YES` and `NO` (in all capitals).

`BOOL` is short for "boolean", after Englishman George Boole who long ago invented a type of logic that forms the basis of all modern computing. The fact that computers talk in ones and zeros is largely due to him. You use `BOOL` variables to remember whether something is true (`YES`) or not (`NO`). The names of boolean variables often start with the verb "is" or "has", as in `isHungry` or `hasIceCream`.

In our case, the ivar `row0checked` is `YES` if the first row has its checkmark set and `NO` if it hasn't. Likewise, `row1checked` reflects whether the second row has a checkmark or not. The same thing goes for the ivars for the other rows.

The delegate method that handles the taps on the rows will now use these new instance variables to determine whether the checkmark for a row needs to be toggled on or off.

» Replace `didSelectRowAtIndexPath` with the following:

ChecklistsViewController.m

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
  UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];

  BOOL isChecked = NO;
  if (indexPath.row == 0) {
    isChecked = row0checked;
    row0checked = !row0checked;
  } else if (indexPath.row == 1) {
    isChecked = row1checked;
    row1checked = !row1checked;
  } else if (indexPath.row == 2) {
    isChecked = row2checked;
    row2checked = !row2checked;
  } else if (indexPath.row == 3) {
```

```
    isChecked = row3checked;
    row3checked = !row3checked;
  } else if (indexPath.row == 4) {
    isChecked = row4checked;
    row4checked = !row4checked;
  }

  if (isChecked) {
    cell.accessoryType = UITableViewCellAccessoryNone;
  } else {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  }

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

We examine `indexPath.row` to find the row in question, and then look up the proper "row checked" instance variable. For the first row that is `row0checked`, for the second row it is `row1checked`, and so on. We store its value into the temporary variable `isChecked`, which we'll use at the bottom of the method to set or remove the checkmark on the cell.

Then we do the following to flip the boolean value around:

```
  row0checked = !row0checked;
```

The `!` symbol is the *logical not* operator. There are a few other logical operators that work on `BOOL` values, such as *and* and *or*, which we'll encounter soon enough. What `!` does is simple: it reverses the meaning of the value. If `row0checked` is `YES`, then `!` makes it `NO`. Conversely, `!NO` is `YES`. Think of `!` as "not": not yes is no and not no is yes. Yes?

» Run the app and observe… that it doesn't work very well. You have to tap a few times on a row to actually make the checkmark go away.

What's wrong here? Simple: if you don't set a value in a `BOOL` variable then its default value is `NO`. So `row0checked` and the others think that there is no checkmark on the row, but the table draws one anyway because we set the checkmark accessory on the prototype cell. In other words: our data model (the "row checked" variables) and the views (the checkmarks inside the cells) are out-of-sync.

There are a few ways we could try to fix this: we could set the `BOOL` variables to `YES` to begin with, or we could remove the checkmark from the prototype cell in the Storyboard editor. Neither is a foolproof solution because what goes wrong here isn't so much that we initialized the "row checked" values wrong or designed the prototype cell wrong, but that we forgot to set the checkmark properly in `cellForRowAtIndexPath`.

When you are asked for a new cell, you always should configure all of its properties. The call to `dequeueReusableCellWithIdentifier` could return a cell that was previously used for a row with a checkmark, so if our row doesn't have a checkmark we have to remove it from the cell at this point (and vice versa). Let's fix that.

» Add the following method above `cellForRowAtIndexPath`:

ChecklistsViewController.m

```
- (void)configureCheckmarkForCell:(UITableViewCell *)cell atIndexPath:(↵
    NSIndexPath *)indexPath
{
  BOOL isChecked = NO;
  if (indexPath.row == 0) {
    isChecked = row0checked;
  } else if (indexPath.row == 1) {
    isChecked = row1checked;
  } else if (indexPath.row == 2) {
    isChecked = row2checked;
  } else if (indexPath.row == 3) {
    isChecked = row3checked;
  } else if (indexPath.row == 4) {
    isChecked = row4checked;
  }

  if (isChecked) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }
}
```

This new method looks at the cell of a certain row (specified by `indexPath`) and makes the checkmark visible if the corresponding "row checked" variable is `YES`, or hides it if `NO`.

We'll call this method in `cellForRowAtIndexPath`, just before we return the cell.

» Change `cellForRowAtIndexPath` to the following (recall that `. . .` means that the existing code at that spot doesn't change):

ChecklistsViewController.m

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  . . .
```

```
  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  return cell;
}
```

» Run the app again.

Now the app works just fine. Initially all the rows are unchecked. Tapping a row checks it, tapping it again unchecks it. The rows and cells are now always in sync. This guarantees that our cell always has the value that corresponds to the row.

Why did I make `configureCheckmarkForCell` a method of its own? Well, we can use it to simplify `didSelectRowAtIndexPath`. As you should know by now that method handles taps on the row and toggles the "row checked" variable and then updates the cell. We can simplify things by letting `configureCheckmarkForCell` do some of the work.

» Change `didSelectRowAtIndexPath` to:

ChecklistsViewController.m

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
  UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];

  if (indexPath.row == 0) {
    row0checked = !row0checked;
  } else if (indexPath.row == 1) {
    row1checked = !row1checked;
  } else if (indexPath.row == 2) {
    row2checked = !row2checked;
  } else if (indexPath.row == 3) {
    row3checked = !row3checked;
  } else if (indexPath.row == 4) {
    row4checked = !row4checked;
  }

  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

» Run the app again and it should still work.

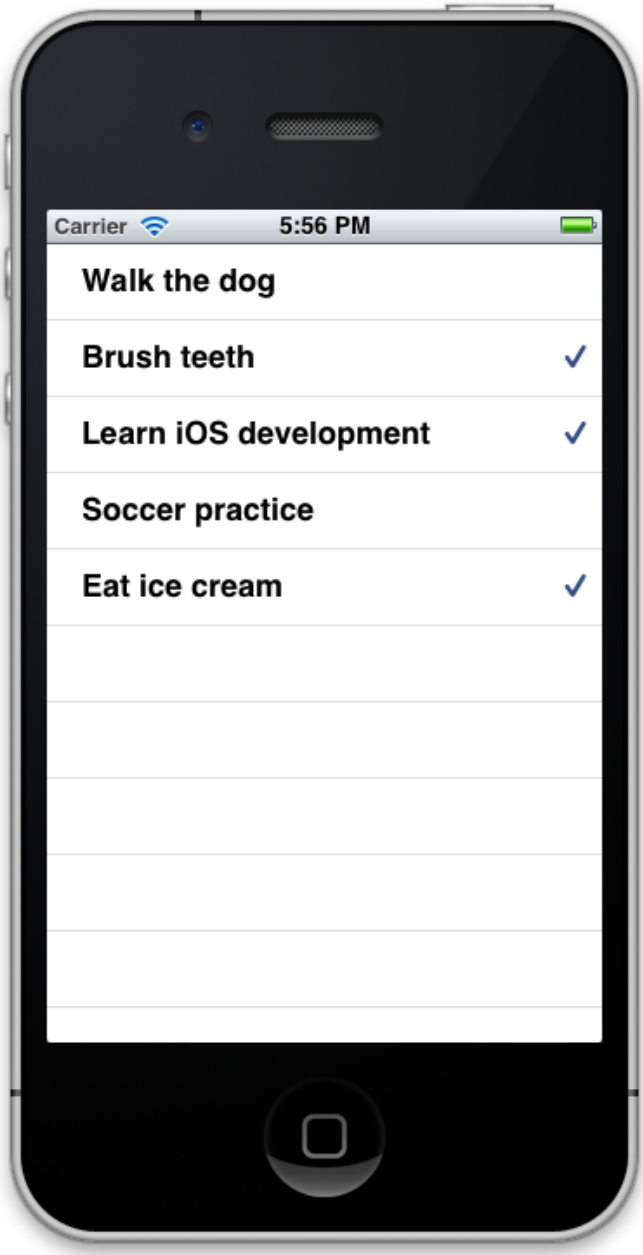
» Add the following to `viewDidLoad` and run the app again:

ChecklistsViewController.m

```
- (void)viewDidLoad
{
  . . .

  row1checked = YES;
  row2checked = YES;
  row4checked = YES;
}
```

Now rows 1, 2 and 4 (i.e the second, third and fifth rows) initially have a checkmark while the others don’t.

The data model and the table view cells are now always in-sync



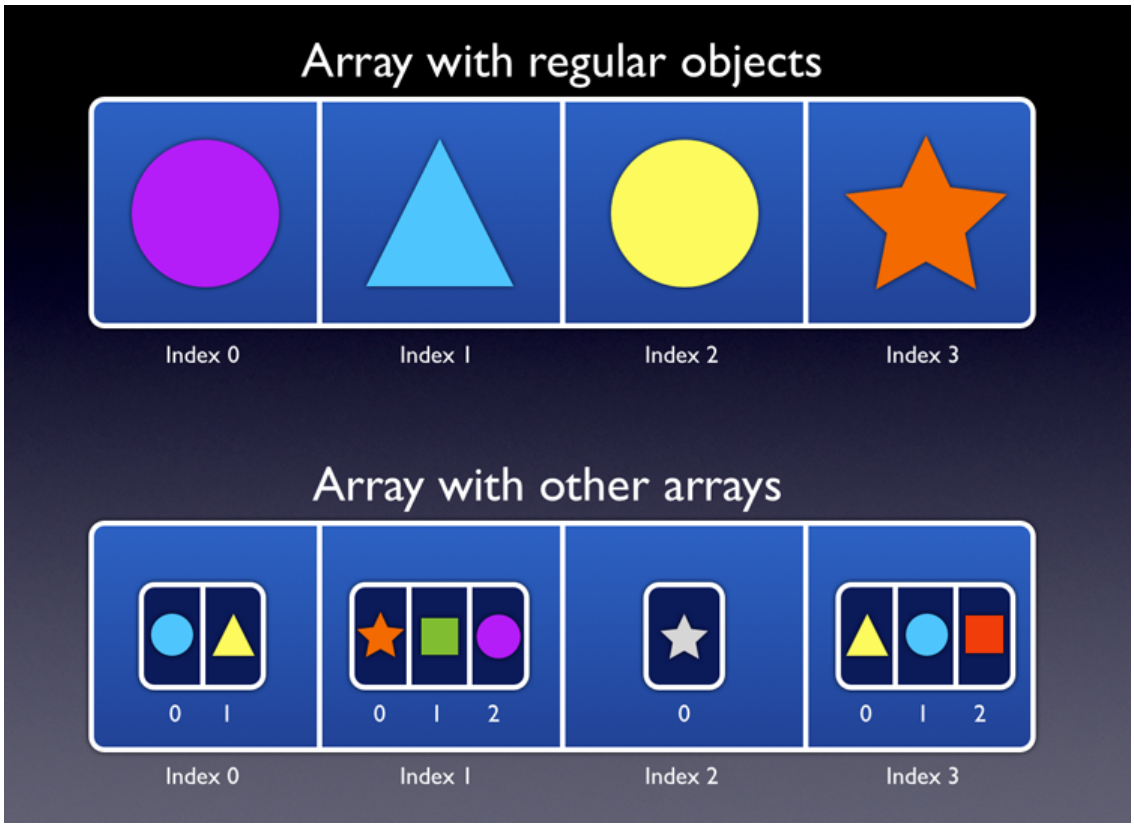
This approach works, but you'll have to agree with me the code is becoming unwieldy very quickly. For only five rows it's doable, but what if we have 100 rows and they all need to be unique? Should we add another 95 "row text" and "row checked" instance variables to the view controller, as well as that many additional if-statements? I hope not!

There is a better way: arrays.

Arrays

An *array* is an ordered list of objects. If you think of a variable as a container of one value (or one object) then an array is a container for multiple objects. Of course, the array itself is also an object (named `NSArray`) that you can put into a variable. And because arrays are objects, arrays can contain other arrays.

Arrays are ordered lists that can contain objects, including other arrays



The objects inside an array are indexed by numbers, starting at 0 as usual. To ask the array for the first object, you do `[array objectAtIndex:0]`. The array is *ordered*, which means that the order of the objects it contains matters. The object at index 0 always comes before the object at index 1.

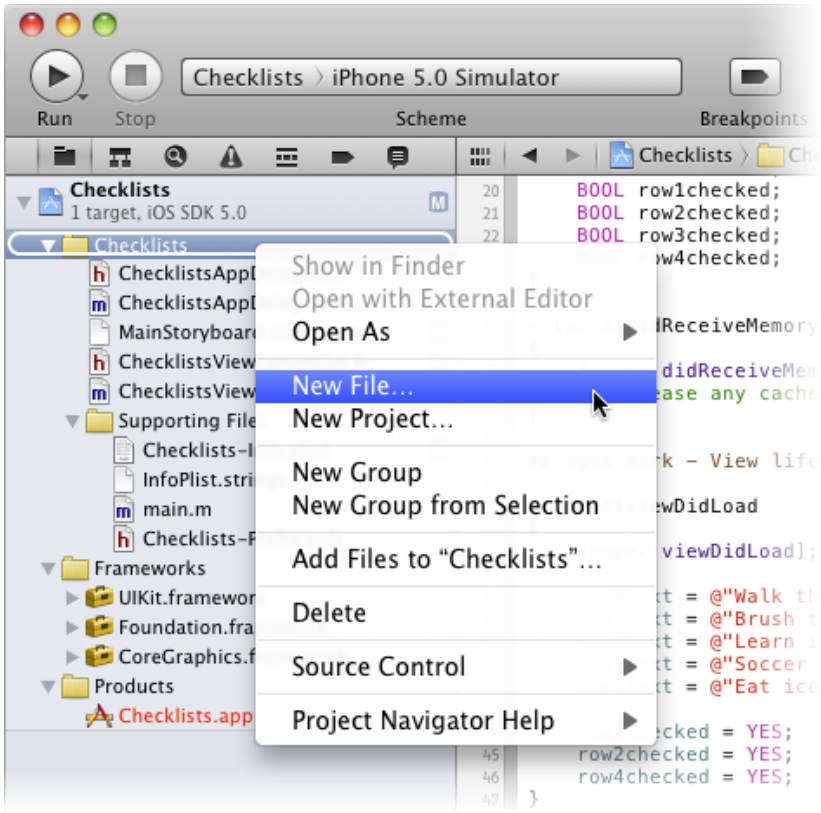
`NSArray` is a so-called *collection* object. There are several other collection objects, such as `NSDictionary` and `NSSet`, and they all organize their objects in a different fashion. (A dictionary contains key-value pairs, just like a real dictionary contains a list of words and a description for each of those words. A set is like an array except that the order of the objects doesn't matter. We'll use these other collections in later tutorials.)

The organization of an array is very similar to the rows from a table — they are both lists of objects in a particular order — so it makes sense that we put our data model's rows into an array.

Arrays store objects, but our rows currently consist of two separate pieces of data: the text and the checked state. It would be easier if we made a single object for each row, because then the row number from the table simply becomes the index in the array. Let's combine the text and checkmark state into a new object of our own!

» Select the Checklists group in the Project Navigator and right click. Choose New File…

Adding a new file to the project

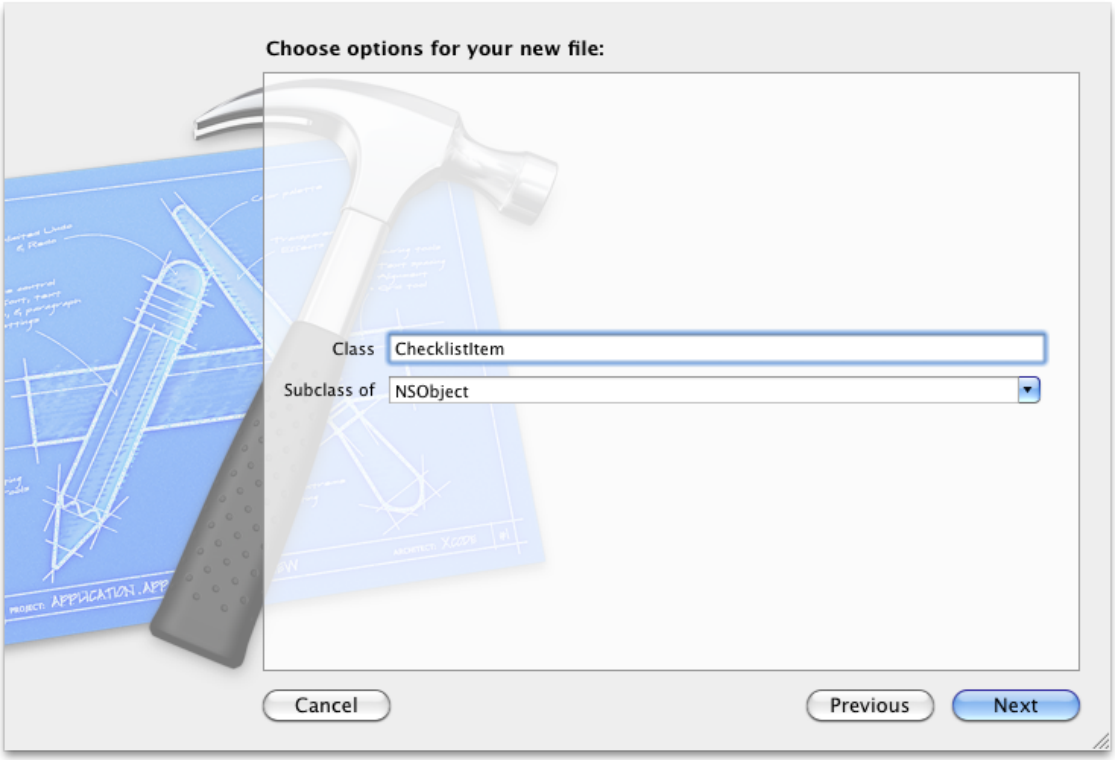


Under the Cocoa Touch section choose Objective-C class:

Choosing the Objective-C class template

The next screen gives you some options to fill out:

Options for the new file

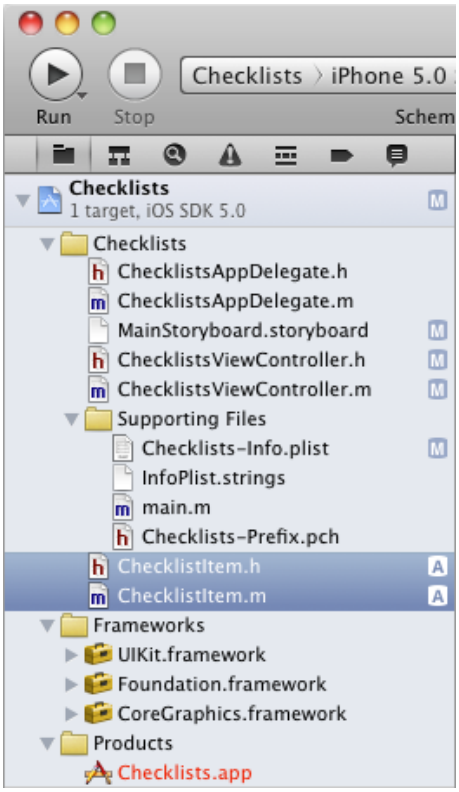


Enter the following:

- Class: `ChecklistItem`
- Subclass of: `NSObject`

On Xcode 4.3 the screen has a few more options, but we will not use them here. Press Next to create the new files, ChecklistItem.h and ChecklistItem.m.

The new files are added to the Project Navigator



The files themselves look like this (without the comments at the top):

ChecklistItem.h

```
#import <Foundation/Foundation.h>

@interface ChecklistItem : NSObject

@end
```

And:

ChecklistItem.m

```
#import "ChecklistItem.h"

@implementation ChecklistItem

- (id)init
{
  self = [super init];
  if (self) {
    // Initialization code here.
  }
```

```
  return self;
}

@end
```

This is roughly the minimum amount of stuff you need in order to make a new object. Xcode added an `init` method to the object already. I'll explain what `init` does in a minute.

Depending on your version of Xcode, ChecklistItem.m may not actually have an `init` method. The developers of Xcode tend to change these templates from time to time, so there may be small differences between what you see in these tutorials and what you see in Xcode. Your ChecklistItem.m may simply look like this:

ChecklistItem.m

```
#import "ChecklistItem.h"

@implementation ChecklistItem

@end
```

If your ChecklistItem.m has no `init` then it will use the standard init method. Because `init` from the Objective-C class template doesn't actually do anything yet, it can be left out. More about these methods later.

» Add the following to ChecklistItem.h, before the `@end` line:

ChecklistItem.h

```
@property (nonatomic, copy) NSString *text;
@property (nonatomic, assign) BOOL checked;
```

These are the two data items that we're adding to the object, in the form of properties. The `text` property will store the description of the checklist item (the text that will appear in the table view cell's label) and the `checked` property determines whether the cell gets a checkmark or not.

Why are we adding these data items as properties and not as instance variables? Instance variables are really supposed to be used on the insides of objects only, they should not be visible to other objects. In this case we do want the text and checked values to be visible. They are part of the `ChecklistItem` object's so-called *public interface* — it is no coincidence that they are placed in the `@interface` section.

Unlike the properties we've used in the previous tutorial, these two do not have the `IBOutlet` symbol as they are not outlets. You only declare something as an outlet when you want to be able to make connections to it from Interface Builder or the Storyboard editor. That is not the case for these properties as they are part of our data model, not the user interface of the app.

As always, we need to `@synthesize` our properties.

» Add the following to ChecklistItem.m below the `@implementation` line:

ChecklistItem.m

```
@synthesize text, checked;
```

That's all for now. The `ChecklistItem` object currently only serves to combine the text and the checked flag into one object.

Before we get around to using an array, let's replace the `NSString` and `BOOL` variables in the view controller with `ChecklistItem` objects.

First, we need to tell the view controller about the `ChecklistItem` object or it won't be able to use it. To do so, we add an `#import` statement to the top of the file.

» Add the following to ChecklistsViewController.m above the `@implementation` line:

ChecklistsViewController.m

```
#import "ChecklistItem.h"
```

» Remove the old `NSString` and `BOOL` instance variables and replace them with `ChecklistItem` objects:

ChecklistsViewController.m

```
@implementation ChecklistsViewController {
  ChecklistItem *row0item;
  ChecklistItem *row1item;
  ChecklistItem *row2item;
  ChecklistItem *row3item;
  ChecklistItem *row4item;
}
```

Previously we filled in the "row text" and "row checked" variables in `viewDidLoad`. We'll do the same for our `ChecklistItem` objects.

» Change `viewDidLoad` to:

ChecklistsViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  rowOitem = [[ChecklistItem alloc] init];
  rowOitem.text = @"Walk the dog";
  rowOitem.checked = NO;

  row1item = [[ChecklistItem alloc] init];
  row1item.text = @"Brush my teeth";
  row1item.checked = YES;

  row2item = [[ChecklistItem alloc] init];
  row2item.text = @"Learn iOS development";
  row2item.checked = YES;

  row3item = [[ChecklistItem alloc] init];
  row3item.text = @"Soccer practice";
  row3item.checked = NO;

  row4item = [[ChecklistItem alloc] init];
  row4item.text = @"Eat ice cream";
  row4item.checked = YES;
}
```

We're essentially doing the same thing as before, except that this time the `text` and `checked` variables are not instance variables of the view controller but properties of the `ChecklistItem` objects.

We begin by creating a new `ChecklistItem` object:

```
  rowOitem = [[ChecklistItem alloc] init];
```

You've seen something similar in the first tutorial when we created the `UIAlertView`. There we did:

```
  alertView = [[UIAlertView alloc] initWithTitle:...];
```

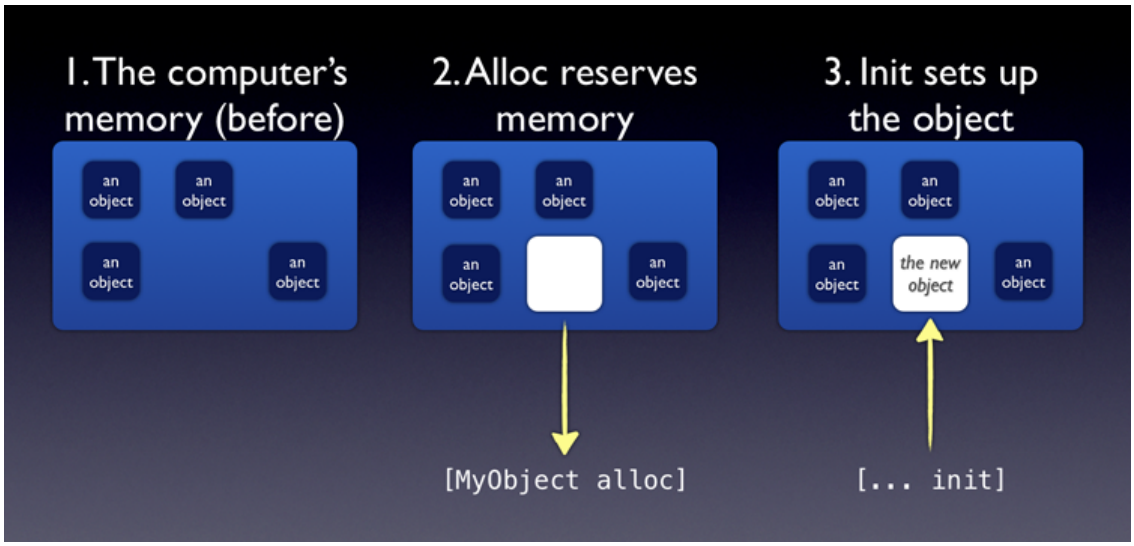
That is how you create objects in Objective-C, you first call `alloc` to reserve memory for this new object, followed by a form of `init` to *initialize* this object. Initialization means

that you put the object in a usable state, usually by giving your instance variables and properties meaningful values.

Our `ChecklistItem` object has an initialization method that is simply named `init`, but not all objects are as concise. The full name of the initialization method for `UIAlertView` is `initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:`. That's quite a mouthful! It's possible for an object to have more than one init method, but you'll always only call one of them (it doesn't make sense to initialize an object twice).

You'll be seeing this pattern a lot, almost every time you make a new object. First `alloc`, then `init`. This gives you an *instance* of the object, a new copy of the object in memory. In case you haven't learned enough fancy words today, the whole process of allocation followed by initialization is also called *instantiation*.

Allocation and initialization



After creating the `ChecklistItem` object, we put values into its `text` and `checked` properties. We repeat this for the four other rows. Each row gets its own `ChecklistItem` object that we store it its own instance variable.

» Change the other methods to:

ChecklistsViewController.m

```
- (void)configureCheckmarkForCell:(UITableViewCell *)cell atIndexPath:(↵
    NSIndexPath *)indexPath
{
  BOOL isChecked = NO;
  if (indexPath.row == 0) {
    isChecked = rowOitem.checked;
  } else if (indexPath.row == 1) {
```

```
    isChecked = row1item.checked;
  } else if (indexPath.row == 2) {
    isChecked = row2item.checked;
  } else if (indexPath.row == 3) {
    isChecked = row3item.checked;
  } else if (indexPath.row == 4) {
    isChecked = row4item.checked;
  }

  if (isChecked) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"↵
      ChecklistItem"];

  UILabel *label = (UILabel *)[cell viewWithTag:1000];

  if (indexPath.row == 0) {
    label.text = row0item.text;
  } else if (indexPath.row == 1) {
    label.text = row1item.text;
  } else if (indexPath.row == 2) {
    label.text = row2item.text;
  } else if (indexPath.row == 3) {
    label.text = row3item.text;
  } else if (indexPath.row == 4) {
    label.text = row4item.text;
  }

  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  return cell;
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
  UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];

  if (indexPath.row == 0) {
    row0item.checked = !row0item.checked;
  } else if (indexPath.row == 1) {
    row1item.checked = !row1item.checked;
  } else if (indexPath.row == 2) {
    row2item.checked = !row2item.checked;
```

```
  } else if (indexPath.row == 3) {
    row3item.checked = !row3item.checked;
  } else if (indexPath.row == 4) {
    row4item.checked = !row4item.checked;
  }

  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Instead of using the `row0text` and `row0checked` variables, we now use `row0item.text` and `row0item.checked`. Likewise for the other rows.

» Run the app just to make sure that everything still works.

The code is still unwieldy because we still need to keep around a `ChecklistItem` instance variable for each row. Time to put that array into action!

Mutable and non-mutable

There are actually two types of arrays: the *mutable* array (`NSMutableArray`) and the *non-mutable* or *immutable* array (`NSArray`). Mutable means: can be changed. An `NSArray`, which is non-mutable, cannot be changed once it is created. You cannot add new objects to it or remove objects from it, only access the objects that are already inside the array.

You see this in other places in the iOS SDK as well. `NSString` is also immutable. Once you've made an `NSString` object, you cannot change its text. You can only create a new string object that is derived from this one. For example, `[string lowercase]` will create a new string with all the characters converted to lowercase. The original string object is still there, unmodified. If you need to create a string that you can change afterwards, you should use the `NSMutableString` object instead.

Note that even if you have an non-mutable array, you can still modify the objects that it contains. It is the array itself that cannot change — you cannot take objects out of it or put new objects into it — but once you have obtained a reference to one of its objects using `[array objectAtIndex:]` you can do with that object what you want.

Think of an immutable array as being stuck in a traffic jam. The cars in front of you and behind you don't change and no one is going anywhere, but you can certainly step out of your car and spray paint it pink.

We need to use a mutable array because in our app we will let the user add new items to the list and remove items as well.

» In ChecklistsViewController.m, throw away all the instance variables and replace them with a single `NSMutableArray` ivar named `items`:

ChecklistsViewController.m

```
@implementation ChecklistsViewController {
  NSMutableArray *items;
}
```

» Change the `viewDidLoad` method to:

ChecklistsViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  items = [[NSMutableArray alloc] initWithCapacity:20];

  ChecklistItem *item;

  item = [[ChecklistItem alloc] init];
  item.text = @"Walk the dog";
  item.checked = NO;
  [items addObject:item];

  item = [[ChecklistItem alloc] init];
  item.text = @"Brush my teeth";
  item.checked = YES;
  [items addObject:item];

  item = [[ChecklistItem alloc] init];
  item.text = @"Learn iOS development";
  item.checked = YES;
  [items addObject:item];

  item = [[ChecklistItem alloc] init];
  item.text = @"Soccer practice";
  item.checked = NO;
  [items addObject:item];

  item = [[ChecklistItem alloc] init];
  item.text = @"Eat ice cream";
  item.checked = YES;
  [items addObject:item];
}
```

This is not so different from before, except that we first make the array object:

```
  items = [[NSMutableArray alloc] initWithCapacity:20];
```

Again, notice the `[[alloc] init...]` pattern to create and initialize the object. It is important to realize that just declaring that you have a variable does not automatically make the corresponding object for you. The variable is just the container for the object. You'll still have to call `alloc` and `init` to create the object and put it into that variable.

When we did this,

```
@implementation ChecklistsViewController {
  NSMutableArray *items;
}
```

we just said: we have a variable named `items` that can contain an `NSMutableArray` object. But until we instantiate an actual `NSMutableArray` object and put that into `items`, the variable is empty. Its value is "nil" in programmer-speak, although some programmers like to call this "null". You can still send messages to a `nil` variable, but they won't arrive anywhere so that's quite pointless.

That's why in `viewDidLoad`, we first make the actual `NSMutableArray` object and stuff it into `items`. Now we can use this array object through the `items` ivar.

`NSMutableArray` has an init method named `initWithCapacity` that reserves space for a certain number of items (20 in our case). That doesn't mean the array can only store 20 items and no more! It's just a hint that we give to the array. We expect about 20 items, but if we add more than that the array will grow to make room.

Each time we make a `ChecklistItem`, we now add it into the array:

```
  item = [[ChecklistItem alloc] init];
  item.text = @"Walk the dog";
  item.checked = NO;
  [items addObject:item];
```

At the end of `viewDidLoad`, the `items` array contains five `ChecklistItem` objects. This is our new data model.

Now that we have all our rows in the `items` array, we can simplify our table view data source and delegate methods.

» Change these methods to:

ChecklistsViewController.m

```
- (void)configureCheckmarkForCell:(UITableViewCell *)cell atIndexPath:(↵
    NSIndexPath *)indexPath
{
  ChecklistItem *item = [items objectAtIndex:indexPath.row];

  if (item.checked) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"↵
      ChecklistItem"];

  ChecklistItem *item = [items objectAtIndex:indexPath.row];

  UILabel *label = (UILabel *)[cell viewWithTag:1000];
  label.text = item.text;

  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  return cell;
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
     *)indexPath
{
  UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];

  ChecklistItem *item = [items objectAtIndex:indexPath.row];
  item.checked = !item.checked;

  [self configureCheckmarkForCell:cell atIndexPath:indexPath];

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

In each method, we do:

```
  ChecklistItem *item = [items objectAtIndex:indexPath.row];
```

This asks the array for the `ChecklistItem` object at the index that corresponds with the row number. Once we have that object, we can simply look at its `text` and `checked`

properties and do whatever we need to do. If we were to add 100 items to this list, then none of this code would need to change. It works equally well with five items as with a hundred.

Speaking of the number of items, we can now change `numberOfRowsInSection` to return the number of items in the array, instead of a hard-coded number.

» Change the `numberOfRowsInSection` method to:

ChecklistsViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
{
  return [items count];
}
```

Not only is the code a lot shorter and easier to read, it can now also handle an arbitrary number of rows. That is the power of arrays.

» Run the app and see for yourself. It should still do exactly the same as before but its internal structure is much better.

Exercise: Add a few more rows to the table. You should only have to change `viewDidLoad` for this to work. ■

Cleaning up the code

There are a few more things I want to do to clean up this code.

» Make these changes:

ChecklistsViewController.m

```
- (void)configureCheckmarkForCell:(UITableViewCell *)cell withChecklistItem:(↵
    ChecklistItem *)item
{
  if (item.checked) {
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
  } else {
    cell.accessoryType = UITableViewCellAccessoryNone;
  }
}

- (void)configureTextForCell:(UITableViewCell *)cell withChecklistItem:(↵
    ChecklistItem *)item
```

```
{
  UILabel *label = (UILabel *)[cell viewWithTag:1000];
  label.text = item.text;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"↵
      ChecklistItem"];

  ChecklistItem *item = [items objectAtIndex:indexPath.row];

  [self configureTextForCell:cell withChecklistItem:item];
  [self configureCheckmarkForCell:cell withChecklistItem:item];

  return cell;
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
  UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];

  ChecklistItem *item = [items objectAtIndex:indexPath.row];
  [item toggleChecked];

  [self configureCheckmarkForCell:cell withChecklistItem:item];

  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Exercise: Spot the differences. Can you see what was changed? Can you reason why? ■

Answer: I renamed the `configureCheckmarkForCell:atIndexPath:` method to `configureCheckmarkForCell:withChecklistItem:`. If you think this is a long method name, you're in for a surprise. A lot of the object names and method names in the iOS SDK are huge, but at least that should give you a good idea of what they mean. Fortunately, Xcode has an auto-completion feature, so you only have to type the first few characters and it will automatically fill out the rest. Otherwise you'd be doing a lot of typing!

Why did I change this method? Previously it received an index-path and then did this to find the corresponding `ChecklistItem`:

```
  ChecklistItem *item = [items objectAtIndex:indexPath.row];
```

But in both `cellForRowAtIndexPath` and `didSelectRowAtIndexPath` we already do that as well. So it makes more sense to pass that `ChecklistItem` object directly to the "configure" method instead of making it do the same work twice. Anything that simplifies the code is good.

I also added a `configureTextForCell:withChecklistItem:` method. That sets the item's text on the cell's label. Previously we did that directly in `cellForRowAtIndexPath` but I thought it would be a little clearer to put that in its own method.

Finally, `didSelectRowAtIndexPath` no longer modifies the `ChecklistItem`'s `checked` property directly but calls a new method named `toggleChecked` on the item object. We still need to add this method to `ChecklistItem` otherwise the code won't run.

» Add the following to ChecklistItem.h, before `@end`:

ChecklistItem.h

```
- (void)toggleChecked;
```

» Add the implementation of this method to ChecklistItem.m:

ChecklistItem.m

```
- (void)toggleChecked
{
  self.checked = !self.checked;
}
```

As you can see, the method does exactly what `didSelectRowAtIndexPath` used to do, except that we've added this bit of functionality to `ChecklistItem` instead. A good object-oriented design principle is that you should let objects change their own state as much as possible. Previously, the view controller implemented this toggling behavior but now `ChecklistItem` knows how to toggle itself.

» Run the app, and well, it still should work exactly the same as before. :-)

If you want to check your work, you can find the project files for the current version of the app in the folder "02 - Arrays" in the tutorial's Source Code folder.

Clean up that mess!

So what's the point of making all of these changes if the app still works exactly the same? For one, the code is much cleaner and that helps to avoid bugs. By using an array we've also made the code more flexible. Our table view can now handle any number of rows.

You'll find that when you are programming you are constantly restructuring your code to make it better. It's impossible to do the whole thing 100% perfect right from the start. So we write code until it becomes messy and then we clean it up. Then after a little while it becomes a big mess again and we clean it up again. This process for cleaning up code is called *refactoring* and it's a cycle that never ends.

There are a lot of programmers who never do clean up their code. The result is what we call "spaghetti code" and it's a horrible mess to maintain. If you haven't looked at your code for several months but then need to add a new feature or fix a bug, you may need some time to read it through to understand again how everything fits together. It's in your own best interest to write code that is as clean as possible, otherwise untangling that spaghetti mess is no fun.

Adding new items to the checklist

So far our table view has contained a handful of fixed rows but the idea behind this app is that you can create your own lists. Therefore, we need to give the user the ability to add their own to-do items.

In this section we'll expand the app to have a so-called *navigation bar* at the top. This bar has an Add button (the big +) that opens new screen that lets you enter a name for the new to-do item. When you tap Done, the new item will be added to the list.

The + button in the navigation bar opens the Add Item screen

Presenting a new screen to add items is a common pattern in a lot of apps. Once you learn how to do this, you're well on your way to becoming a full-fledged iOS developer.

What we'll do in this section:

- Add a navigation controller
- Put the Add button into the navigation bar
- Add a fake item to the list when you press the Add button
- Delete items with swipe-to-delete
- Open the Add Item screen that lets the user type the text for the item

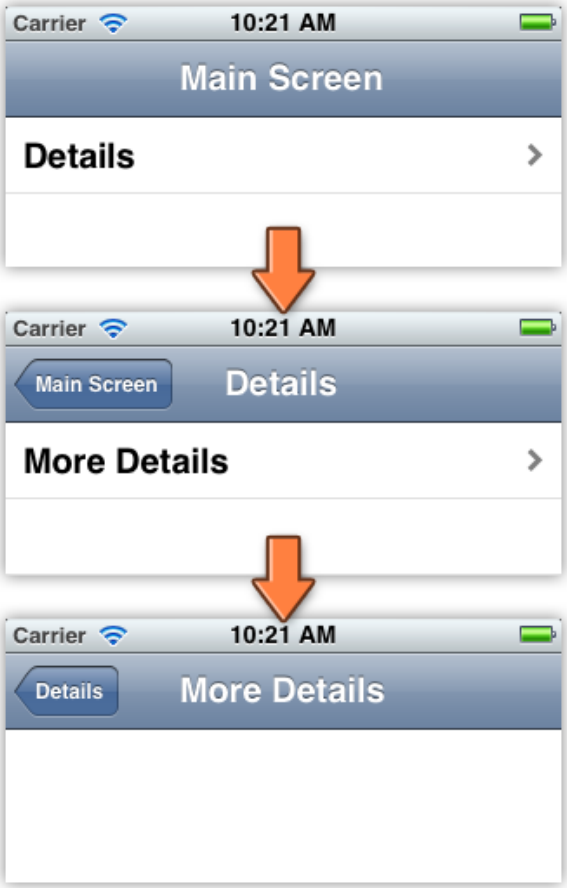
As always, we take it in small steps. After we've put the Add button on the screen, we first write the code to add a "fake" item to the list. Instead of writing all of the code for the Add Item screen, we simply pretend that it already exists. Once we've learned how to add fake items, we can build the Add Item screen for real.

Navigation controllers

First, let's add the navigation bar. You may have seen in the Object Library that there actually is an object named Navigation Bar. You can drag this into your view and put it at the top. However, we won't do that. Instead, we will embed our view controller inside a *navigation controller.*

Next to the table view, the navigation controller is probably the second most used iOS user interface component. It is the thing that lets you go from one page to another:

A navigation controller in action

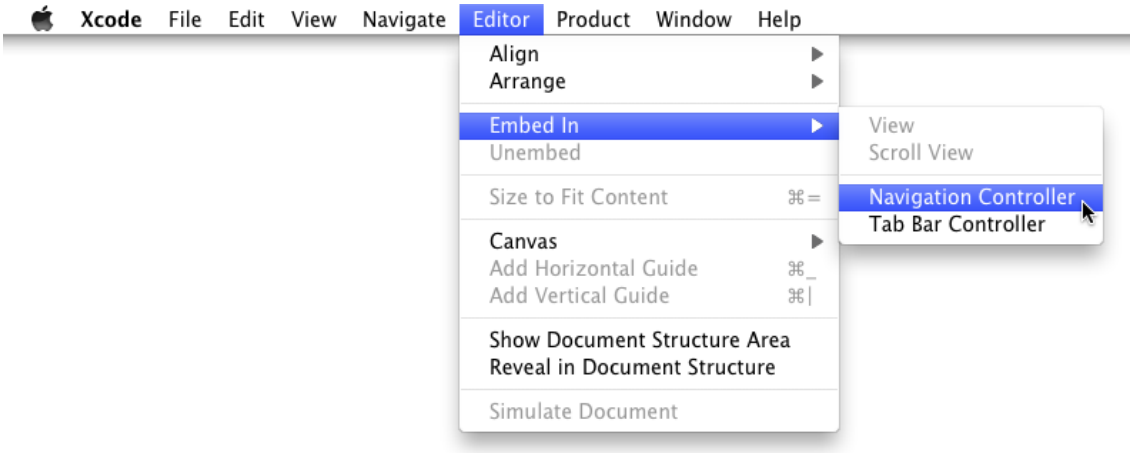


The `UINavigationController` takes care of most of this navigation stuff for you, which saves a lot of programming effort. You get a title in the middle of the screen and a "back" button that automatically takes the user back to the previous screen. You can put a button of your own on the right.

Adding a navigation controller is really easy.

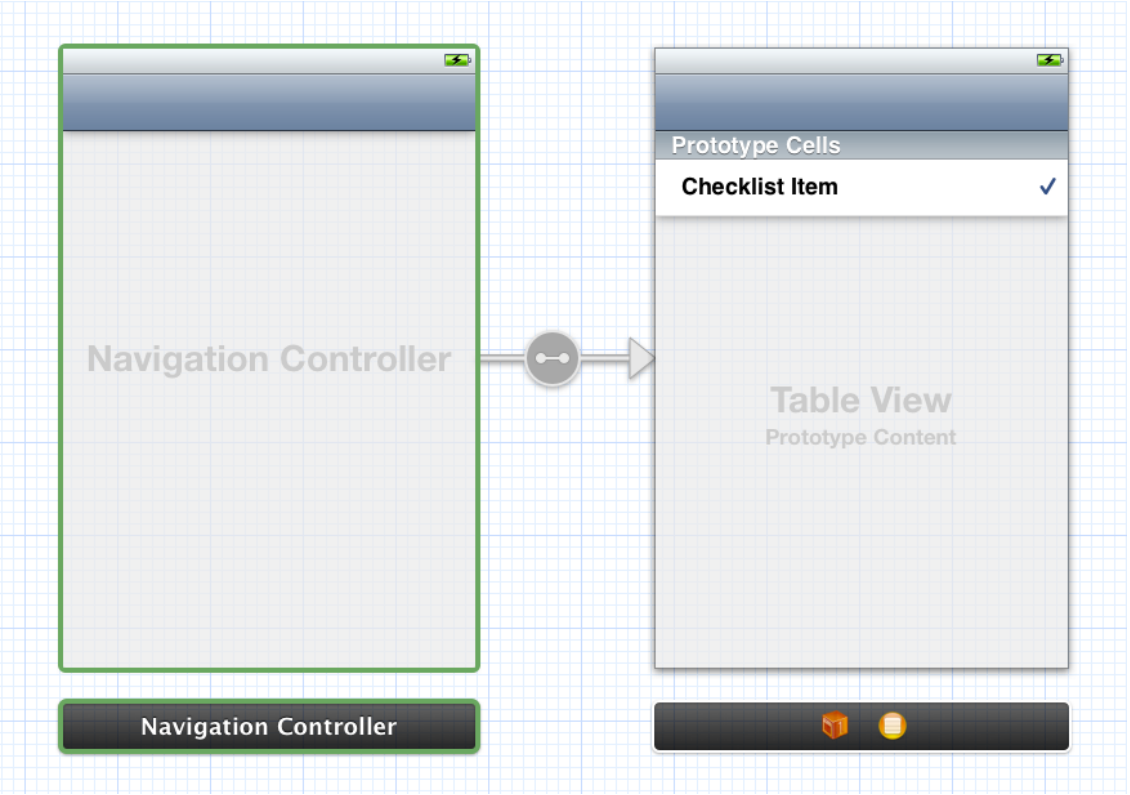
» Go to the Storyboard editor and select the Checklists View Controller scene. From the menu bar at the top of the screen, choose Editor → Embed In → Navigation Controller.

Putting the view controller inside a navigation controller



That's it. The Storyboard editor has now added a new Navigation Controller scene and made a relationship between it and our view controller.

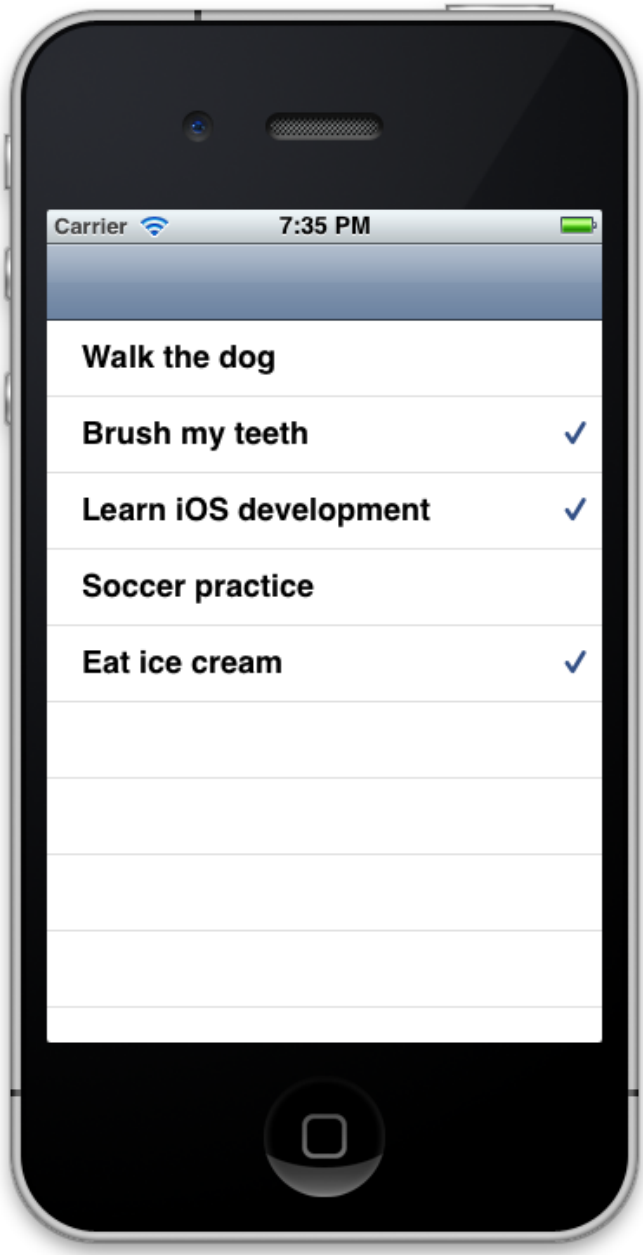
The navigation controller is now linked with our view controller



When the app starts up, the Checklists View Controller is automatically put inside a navigation controller.

» Run the app and try it out.

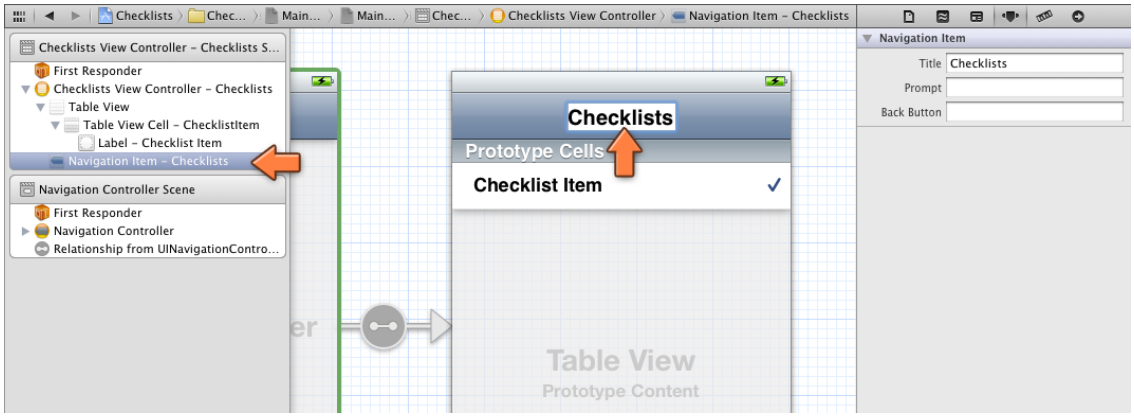
The app now has a navigation bar at the top



The only thing different (visually) is that we now have a navigation bar at the top.

» Go back to the Storyboard editor and double-click on the navigation bar inside the Checklists View Controller to make the title editable. Name it “Checklists”.

Changing the title in the navigation bar

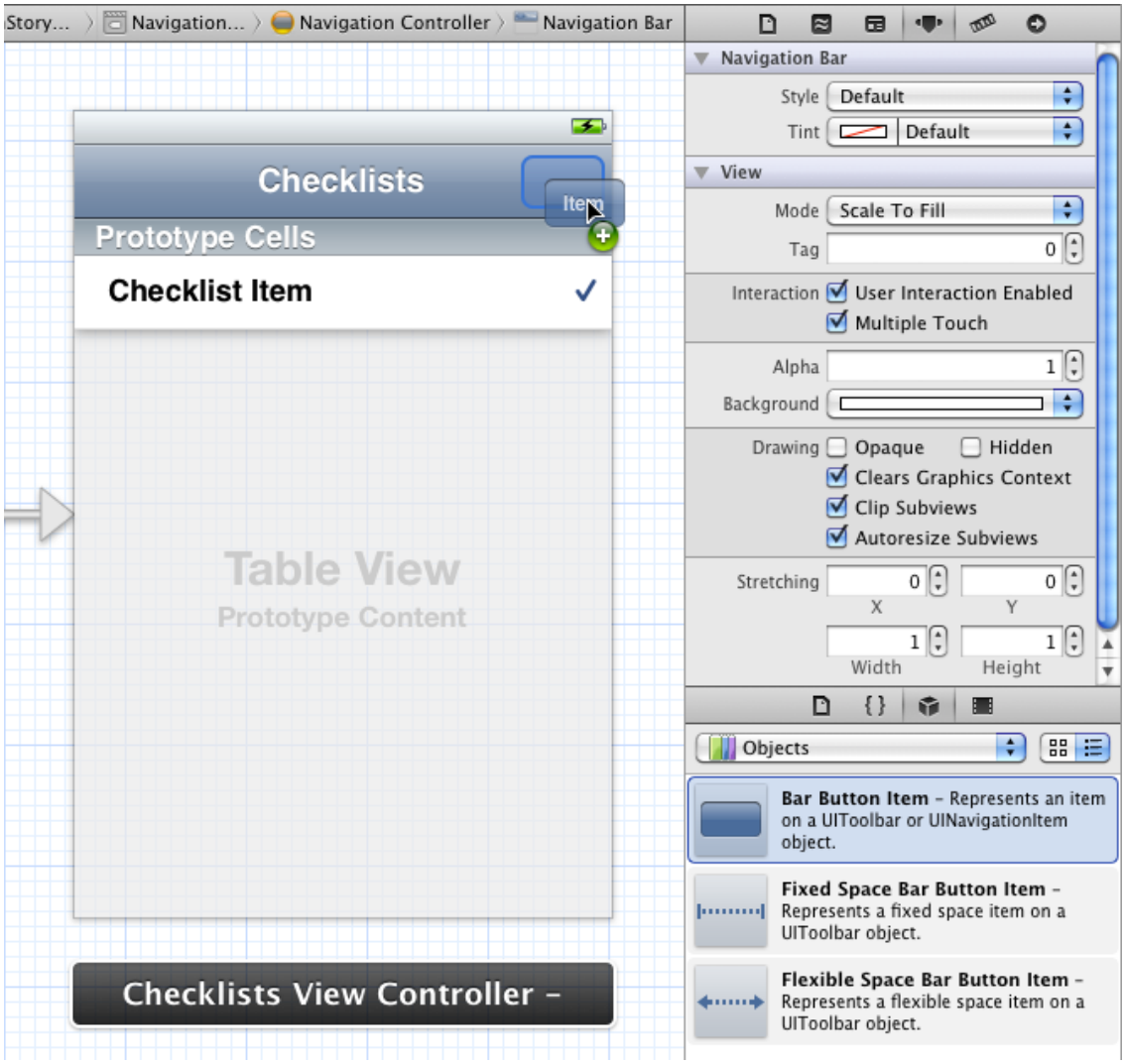


What you're doing here, is changing a *navigation item* object that was automatically added to the view controller when we chose the Embed In command. The Navigation Item object contains the title and buttons that will appear in the navigation bar when this view controller becomes active.

Each embedded view controller has its own Navigation Item that it uses to configure what is inside the navigation bar. When the navigation controller slides a new view controller into the screen, it replaces the contents of the navigation bar with that view controller's Navigation Item.

» Go to the Object Library and look for Bar Button Item. Drag it into the right-side slot of the navigation bar. Be sure to use the navigation bar on the Checklists View Controller, not the one from the navigation controller!

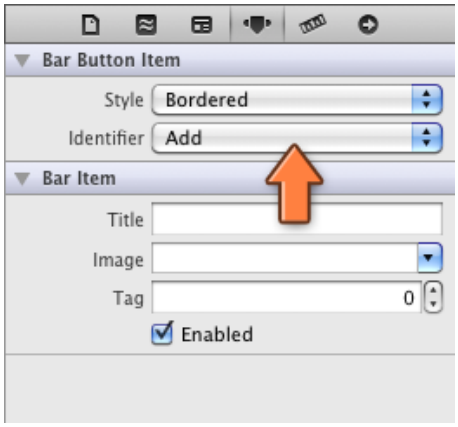
Dragging a Bar Button Item into the navigation bar



By default this new button is named “Item” but we want it to have a big + sign.

» In the Attributes Inspector, choose Identifier: Add. This gives the button the + sign.

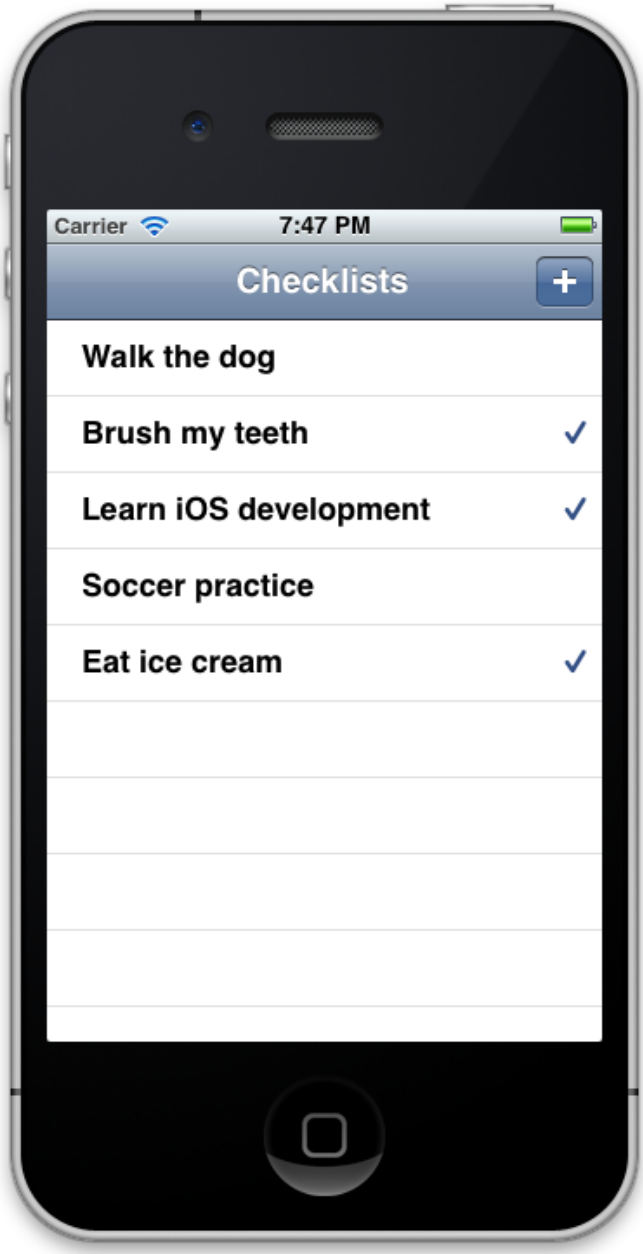
Bar Button Item attributes



If you look through the Identifier list you see a lot of predefined bar button types: Add, Compose, Reply, Camera, and so on. You can use these in your own apps but only for their intended purpose. You shouldn't use the camera icon on a button that sends an email, for example. Improper use of these icons may lead Apple to reject your app from the App Store and that sucks.

OK, that gives us a button. If you run the app, it should look like this:

The app with the Add button



Of course, pressing the button doesn't actually do anything because we haven't hooked it up to an action yet. In a little while we will create the Add Item screen and show this screen when you tap the button. But before we can do that, we'll first have to figure out how to add new rows to the table.

Let's hook up the Add button to an action. You got plenty of exercise on this in the previous tutorial, so this shouldn't be too much of a problem.

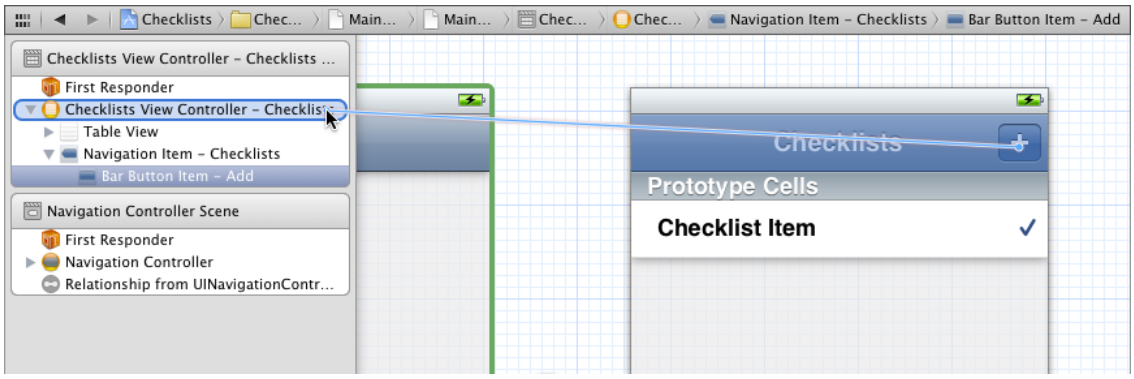
» Add a declaration for a new action method to ChecklistsViewController.h, before `@end`:

ChecklistsViewController.h

```
- (IBAction)addItem;
```

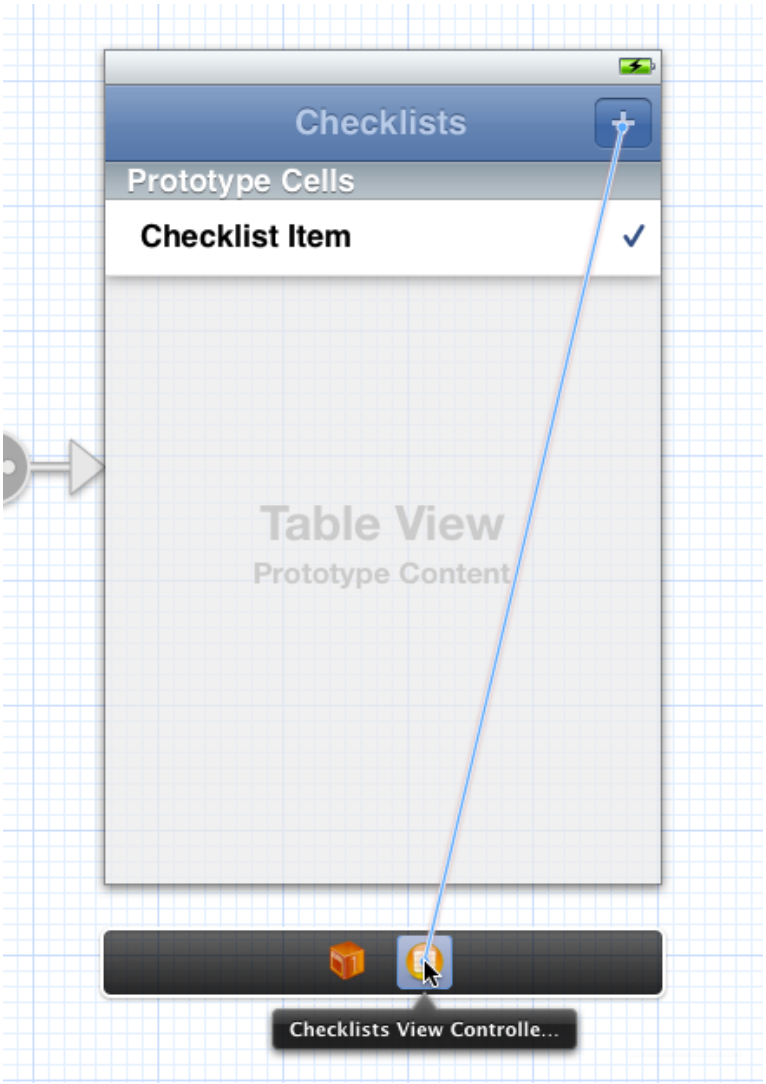
» Go to the Storyboard editor and hook up the Add button to this action. This works as before when we edited nibs, although in the Storyboard editor the scenes do not have a File's Owner. Instead, you can Ctrl-drag to the Checklists View Controller item in the sidebar:

Ctrl-drag from Add button to View Controller



Or, even simpler, Ctrl-drag from the Add button to the view controller item in the dock area below the scene:

Ctrl-drag from Add button to View Controller (alternative method)



In fact, you can Ctrl-drag from the Add button to almost anywhere into the same scene to make the connection (dragging onto the status bar is a good spot).

» After dragging, pick “addItem” from the list (under Sent Actions). Now the connection is made and a tap on the + button will send the `addItem` message to the view controller.

Let’s give `addItem` something to do.

» Add the body of this new method to the bottom of ChecklistsViewController.m, just before `@end`:

ChecklistsViewController.m

```
- (IBAction)addItem
```

```
{
  int newRowIndex = [items count];

  ChecklistItem *item = [[ChecklistItem alloc] init];
  item.text = @"I am a new row";
  item.checked = NO;
  [items addObject:item];

  NSIndexPath *indexPath = [NSIndexPath indexPathForRow:newRowIndex inSection↵
      :0];
  NSArray *indexPaths = [NSArray arrayWithObject:indexPath];
  [self.tableView insertRowsAtIndexPaths:indexPaths withRowAnimation:↵
      UITableViewRowAnimationAutomatic];
}
```

Inside this method we create a new `ChecklistItem` object and add it to our data model. We also have to figure out the row number of this new object and then tell the table view, "We've inserted a row at this index, please update yourself."

Let's take it section by section:

```
int newRowIndex = [items count];
```

When you start the app there are 5 items in the array and 5 rows on the screen. Computers start counting at 0, so the existing rows have indexes 0, 1, 2, 3 and 4. We will add the new row to the end of the array, so the index for that new row will be 5.

In other words, when we're adding a row to the end of a table the index for the new row is always equal to the number of items currently in that table. Let that sink in for a second. We put the index for the new row in the local variable `newRowIndex`.

The following few lines should look familiar:

```
ChecklistItem *item = [[ChecklistItem alloc] init];
item.text = @"I am a new row";
item.checked = NO;
[items addObject:item];
```

You have seen this code before in `viewDidLoad`. It creates the new `ChecklistItem` object and adds it to the end of the array. Our data model now consists of 6 `ChecklistItem` objects inside the `items` array. Note that `newRowIndex` is still 5 even though `[items count]` is now 6. That's why we read the item count and stored this value in `newRowIndex` before we added the new item to the array.

Here it gets tricky:

```
NSIndexPath *indexPath = [NSIndexPath indexPathForRow:newRowIndex inSection:0];
```

Just adding the new `ChecklistItem` object to the data model isn't enough. We also have to tell the table view about this new row so it can add a new cell for that row. As you know by now, table views use index-paths to identify rows, so first we make an `NSIndexPath` object that points to our new row, using the row number from the `newRowIndex` variable. This index-path object now points to row 5 (in section 0).

```
NSArray *indexPaths = [NSArray arrayWithObject:indexPath];
```

We will use the table view method `insertRowsAtIndexPaths` to tell it about the new row, but as its name implies this method actually lets you insert multiple rows at the same time. Instead of a single `NSIndexPath` object, we need to give it an array of index-paths. Not very convenient, but that's the way it is. Fortunately it is easy to create an array that contains a single index-path object using `[NSArray arrayWithObject]`.

Finally, we tell the table view to insert this new row with a nice animation:

```
[self.tableView insertRowsAtIndexPaths:indexPaths withRowAnimation:↵
    UITableViewRowAnimationAutomatic];
```

To recap, we 1) created a new `ChecklistItem` object, 2) added it to our data model, and 3) inserted a new cell for it in the table view.

» Try it out. You can now add many new rows to the table. You can also tap these new rows to turn their checkmarks on and off again. When you scroll the table up and down, the checkmarks stay with the proper rows.

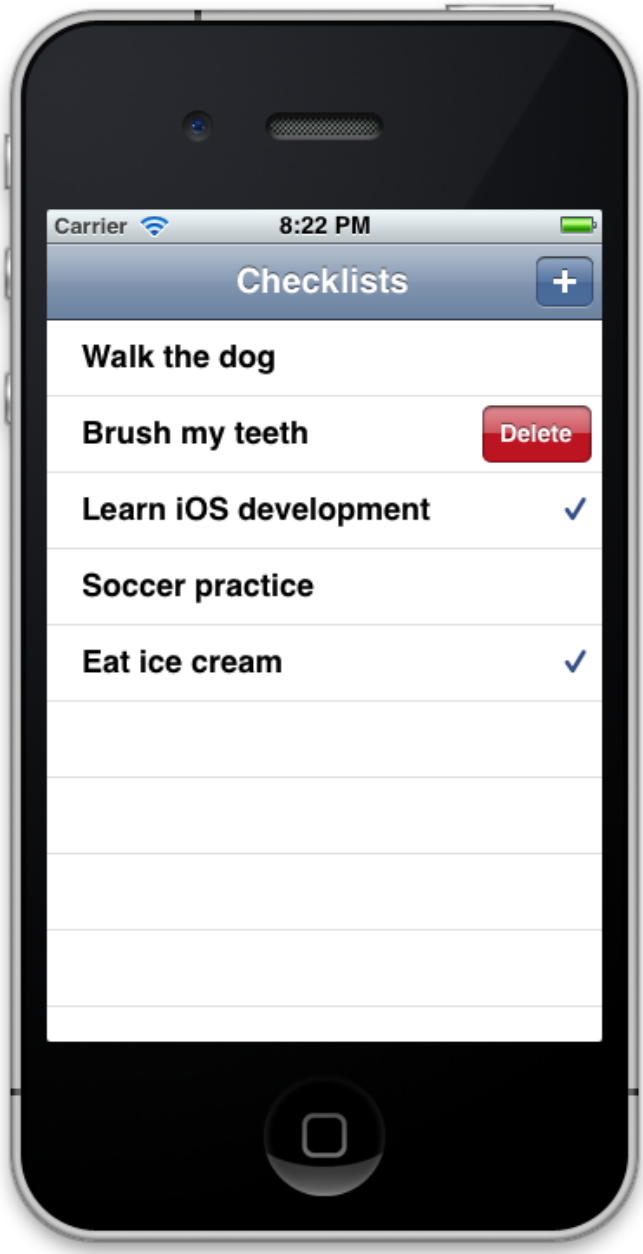
Note that the rows are always added to both the table and our data model. When we send the `insertRowsAtIndexPaths` message to the table, we say: "Hey table, our data model has a bunch of new items added to it." This is important! If you forget to tell the table view about your new items or if you tell the table view there are new items but you don't actually add them to your data model, then your app will crash. These two things always have to be in sync.

Exercise: Give the new items checkmarks by default. ■

Deleting rows

While we're at it, we might as well give users the ability to delete rows. A common way to do this in iOS apps is "swipe-to-delete". You swipe your finger over a row and a Delete button slides into the screen. You then tap the Delete button to confirm the removal or anywhere else to cancel.

Swipe-to-delete in action



Swipe-to-delete is very easy to implement.

» Add the following method to the bottom of ChecklistsViewController.m, before `@end`:

ChecklistsViewController.m

```
- (void)tableView:(UITableView *)tableView commitEditingStyle:(↵
    UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)↵
    indexPath
{
  [items removeObjectAtIndex:indexPath.row];

  NSArray *indexPaths = [NSArray arrayWithObject:indexPath];
  [tableView deleteRowsAtIndexPaths:indexPaths withRowAnimation:↵
      UITableViewRowAnimationAutomatic];
}
```

When the `commitEditingStyle` method is present (it comes from the table view data source), the table view will automatically enable swipe-to-delete. All we have to do is remove the item from our data model:

```
[items removeObjectAtIndex:indexPath.row];
```

And delete the corresponding row from the table view:

```
NSArray *indexPaths = [NSArray arrayWithObject:indexPath];
[tableView deleteRowsAtIndexPaths:indexPaths withRowAnimation:↵
    UITableViewRowAnimationAutomatic];
```

This mirrors what we did in `addItem`. Again we make an `NSArray` with only one index-path object and then tell the table view to remove the rows with an animation.

If at any point you got stuck, you can refer to the project files for the app from the “03 - Data Model” folder in the tutorial’s Source Code folder.

Destroying objects

By the way, when we do `[items removeObjectAtIndex]`, that not only takes the `ChecklistItem` at that index out of the array but it also permanently destroys that `ChecklistItem` object.

We'll talk more about this in the next tutorial, but if there are no more references to an object, it is automatically destroyed. As long as a `ChecklistItem` object sits inside an array, that array has a reference to it. But when we pull the `ChecklistItem` out of the array, that reference goes away and the object is *deallocated*.

What does it mean for an object to be destroyed? Each object occupies a small section of the computer's memory. When you call `alloc` to create an object, a chunk of memory is reserved to hold the object's values. If the object is deallocated, that memory becomes available again and will eventually be occupied by new objects. After it has been deleted, the object is not valid anymore and you can no longer use it.

On versions of iOS before 5.0 you had to take care of this memory management by hand and if you made a mistake it was possible to keep using an object that already had been deleted. This so-called zombie object is no longer valid but you're still trying to access the memory that used to be reserved for it. Sometimes it even works — which is what makes these kinds of bugs so insidious — but eventually your app will crash. With iOS 5 it's a lot harder to use such undead objects, but not impossible.

The Add Item screen

You've learned how to add new rows to the table, but all of these rows get the same text. We will now change the `addItem` action to open a new screen that lets the user enter his own text for those new `ChecklistItems`.

The to-do list for this section:

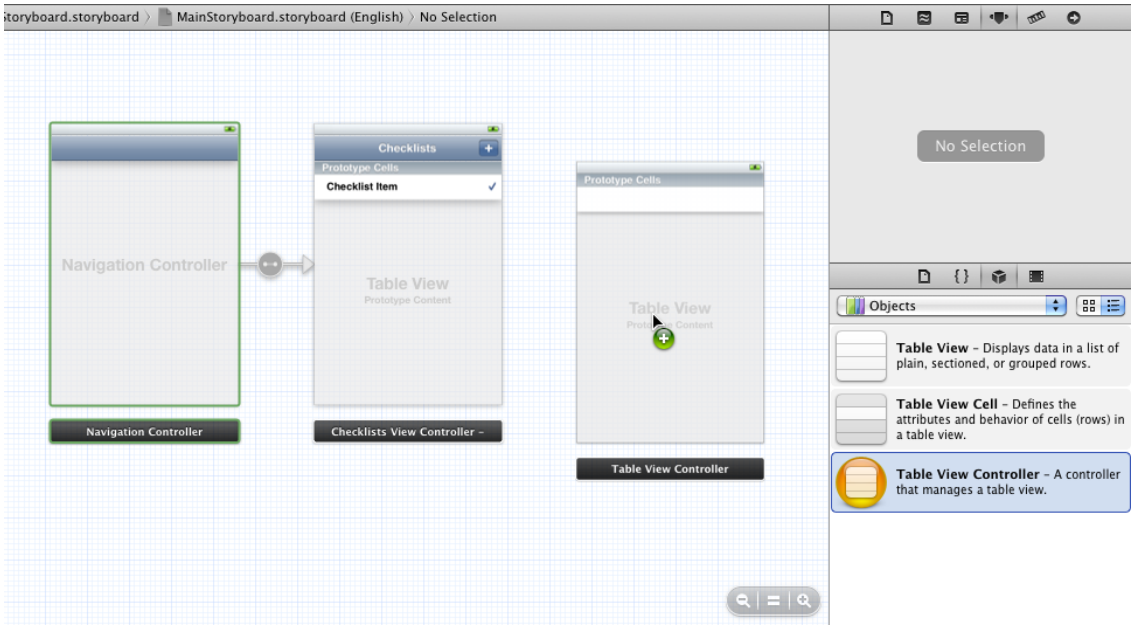
- Create the Add Item screen using the power of Storyboarding
- Add a text field and allow the user to type into it using the on-screen keyboard
- Recognize when the user presses Cancel or Done on the Add Item screen
- Create a new `ChecklistItem` with the text from the text field
- Add the new `ChecklistItem` object to the table on the main screen

A new screen means a new view controller, so we begin by adding a new scene to the Storyboard.

» Go to the Objects Library and drag a new Table View Controller (not a regular view controller) into the Storyboard canvas.

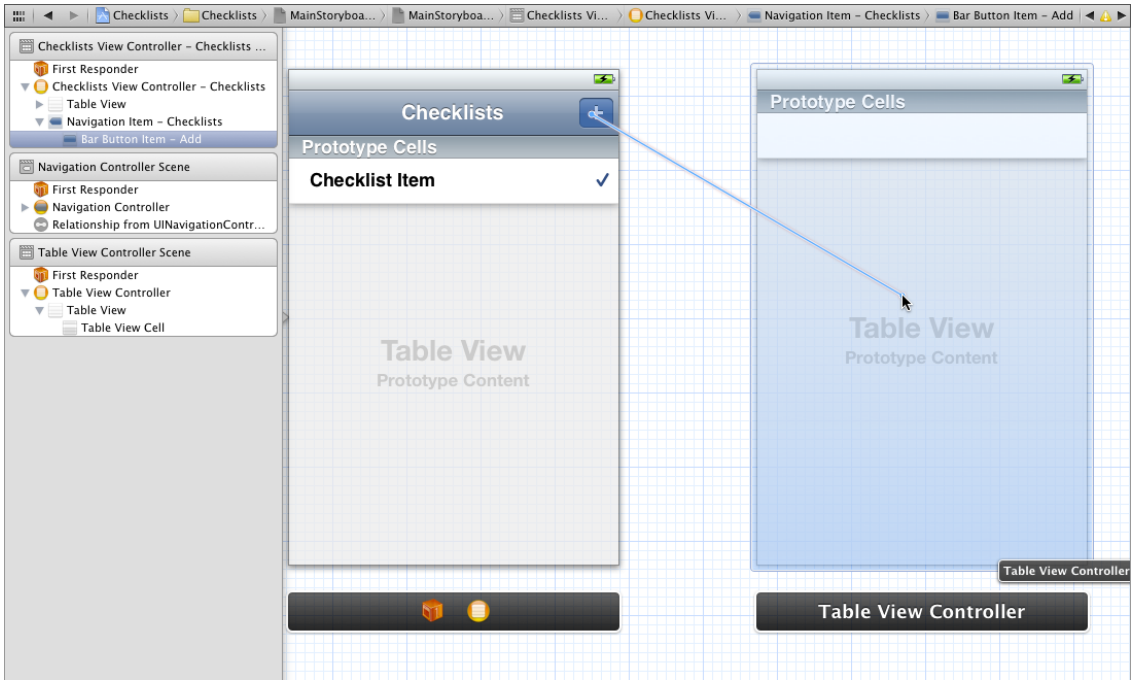
You may need to zoom out to fit everything properly. Either use the loupe icons at the bottom of the screen or double-click the mouse on the canvas.

Dragging a new Table View Controller into the canvas



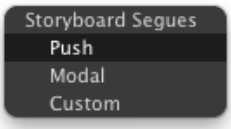
» With the new view controller in place, zoom back in and select the Add button from the Checklists View Controller. Ctrl-drag to the new view controller.

Ctrl-drag from the Add button to the new table view controller



Let go of the mouse and a menu named “Storyboard Segues” pops up:

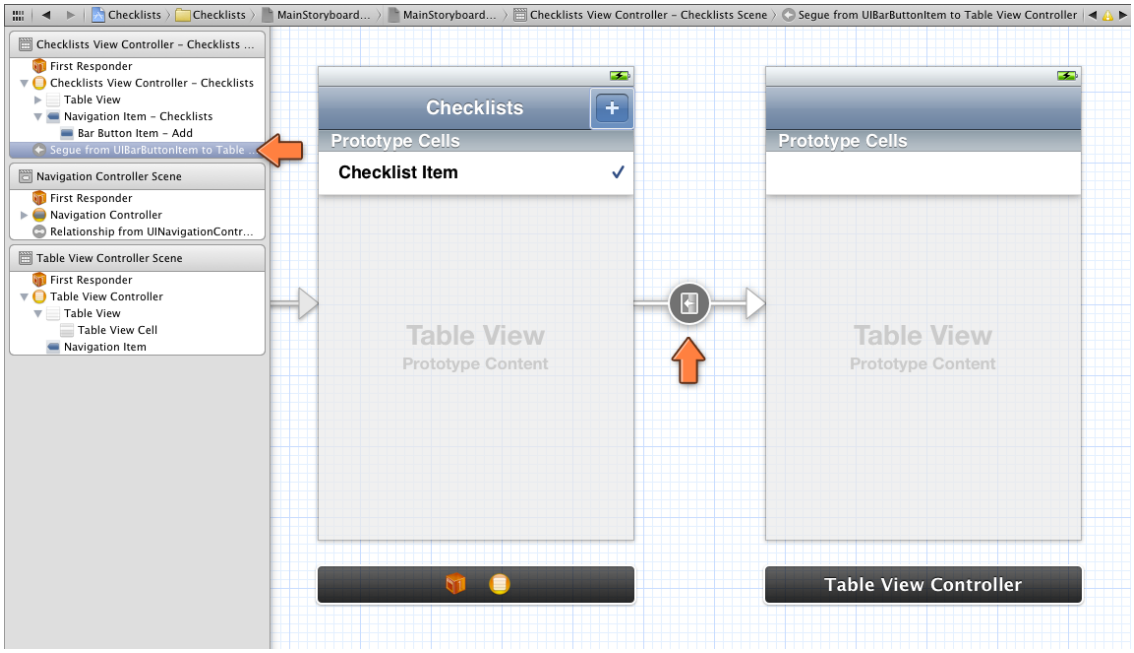
The Storyboard Segues popup



The three options in this menu are the different types of connections you can make between the Add button and the new screen. Choose Push from the menu.

This type of connection is named a *segue* (if you’re not a native English speaker, that is pronounced “seg way” like the strange scooters that you can stand on).

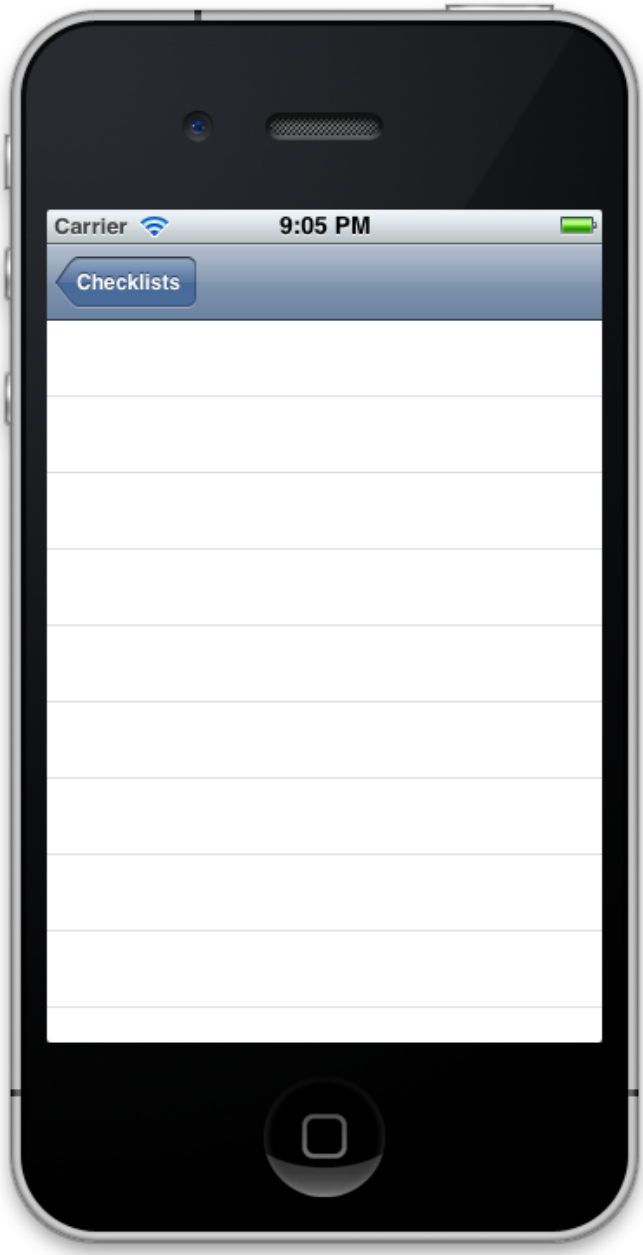
A new segue is added between the two view controllers



» Run the app to see what it does.

When you press the Add button, a new empty table slides in from the right. You can press the Checklists button at the top to go back to the previous screen.

The screen that shows up after you press the Add button



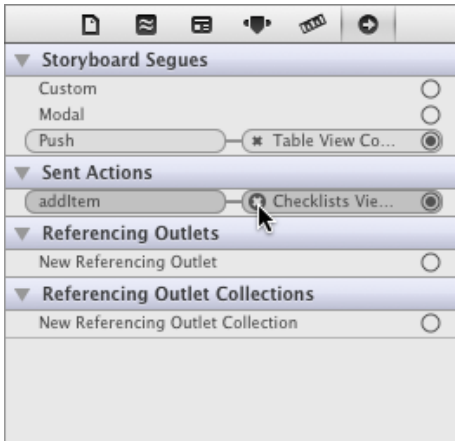
We didn't even have to write any code and we already have ourselves a working navigation controller!

Note that the Add button no longer adds a new row to the table. That connection has been broken and is replaced by the segue. Just in case, we should remove the button's connection with the `addItem` action.

» Select the Add button, go to the Connections Inspector, and press the small X next to

addItem.

Removing the addItem action from the Add button

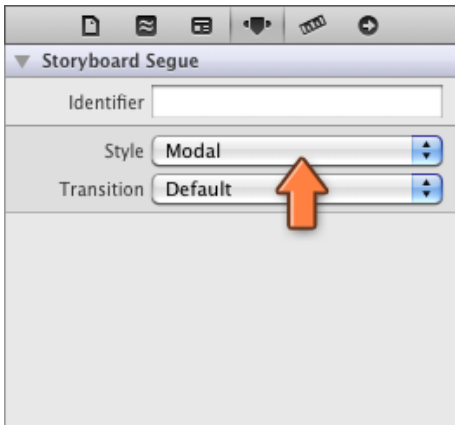


Notice that the Inspector also shows the connection with the segue that we've just made (under Storyboard Segues).

So now we have a new table view controller that slides into the screen when you press the Add button. This isn't actually what we want, though. For a screen that lets you add new items, it is better to use a so-called *modal* segue.

» Click the segue in the Storyboard editor. The segue is an object like any other (remember, everything is an object!) and as such it has attributes that you can change. In the Attributes Inspector, choose Style: Modal.

Changing the segue style to Modal



The navigation bar now disappears from the new view controller. This new screen is no longer presented as part of the navigation hierarchy, but as a separate screen that lies on top of the existing one.

» Run the app to see the difference.

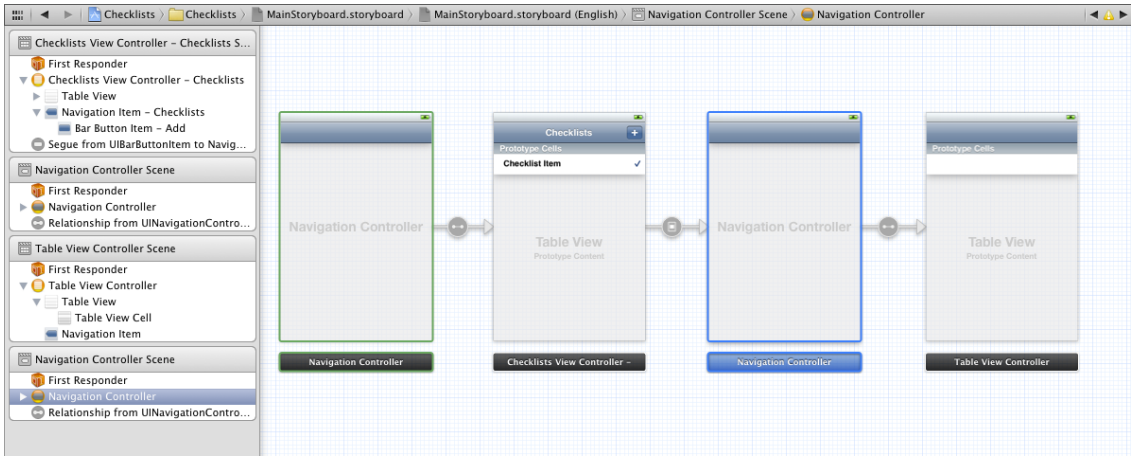
When you do, you'll notice that you no longer have a way to go back to the previous screen. Eek! Here's what we'll do: modal screens usually have a navigation bar with a Cancel button on the left and a Done button on the right. (In some apps the button on the right is called Save or Send.) Pressing either of these buttons will close the screen, but only Done will save your changes.

The easiest way to add a navigation bar and two buttons is to wrap the view controller for the Add Item screen into a navigation controller of its own. The steps to do this are the same as before:

» Select the table view controller (the new one), choose Editor → Embed In → Navigation Controller.

Now the storyboard looks like this:

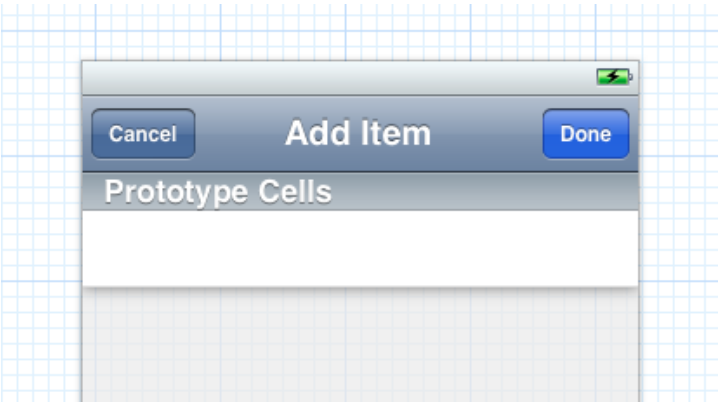
The Storyboard with our two table view controllers that are both embedded in their own navigation controller



The new navigation controller has been inserted in between the two table view controllers. The Add button now performs a modal segue to the new navigation controller.

» Double-click the navigation bar in the right-most view controller to edit its title and change it to "Add Item". Drag two Bar Button Items into the navigation bar, one in the left slot and one in the right slot.

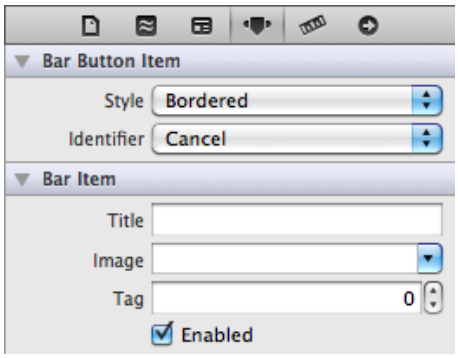
The navigation bar items for the new screen



» In the Attributes Inspector for the left button choose Identifier: Cancel. For the right button choose Identifier: Done.

You don't have to type anything into the button's Title field. The Cancel and Done buttons are built-in button types that automatically use the proper text.

Cancel and Done are built-in button types



You could also choose Identifier: Custom and type the text "Cancel" or "Done" into the button. However, there is an advantage to choosing these predefined Cancel and Done buttons. For one, the Done button has a slightly different background color that you don't get when you use a custom item. Two, if your app runs on an iPhone where the language is set to something other than English, the standard buttons are automatically translated into the user's language.

» Run the app and you'll see that our new screen has Cancel and Done buttons.

Making our own view controller object

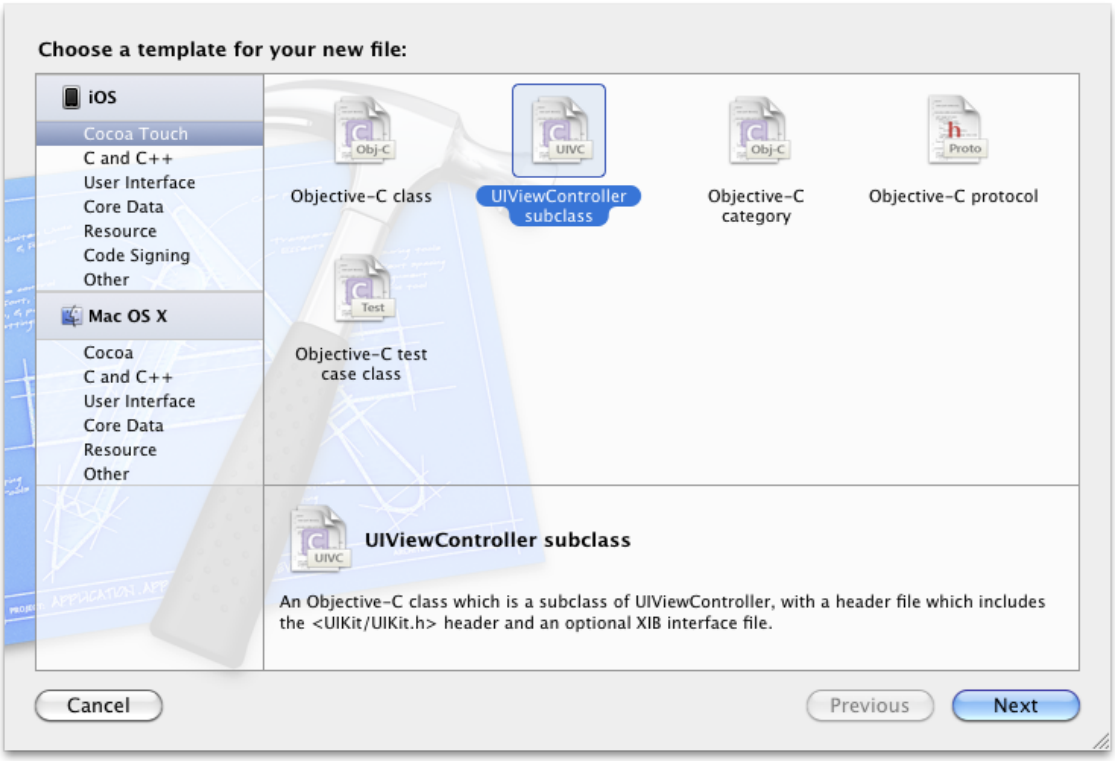
The Cancel and Done buttons should close the Add Item screen and return the app to the main screen. There is currently no way to perform this kind of "backwards" segue from

the Storyboard editor, so we'll have to write the code for that ourselves — in other words, we have to hook up these buttons to action methods.

Where do we put these action methods? Not in `ChecklistsViewController` because that is not the view controller we're dealing with here. Instead, we have to make a new view controller object specifically for the Add Item screen and connect it to the scene we've just designed in the Storyboard editor.

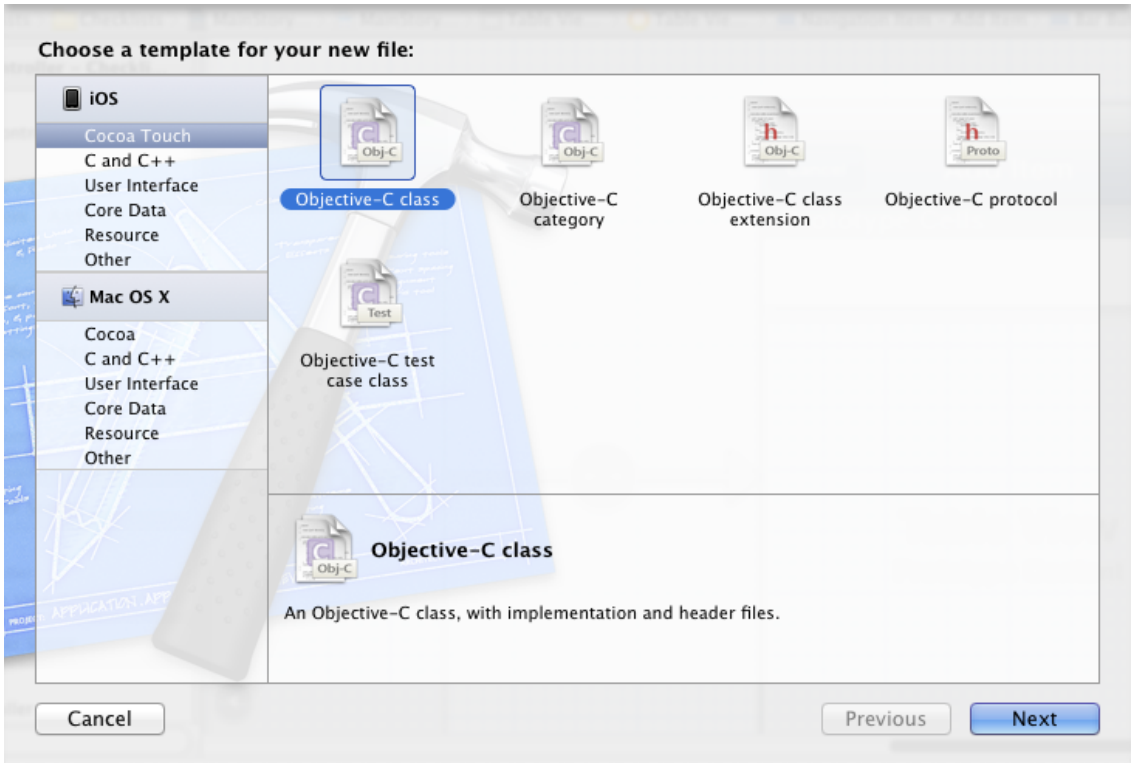
» Right-click on the Checklists group in the Project Navigator and choose New File… This time choose the "UIViewController subclass" template.

Choosing the UIViewController subclass template



In Xcode 4.3, the templates are different. There you choose "Objective-C class":

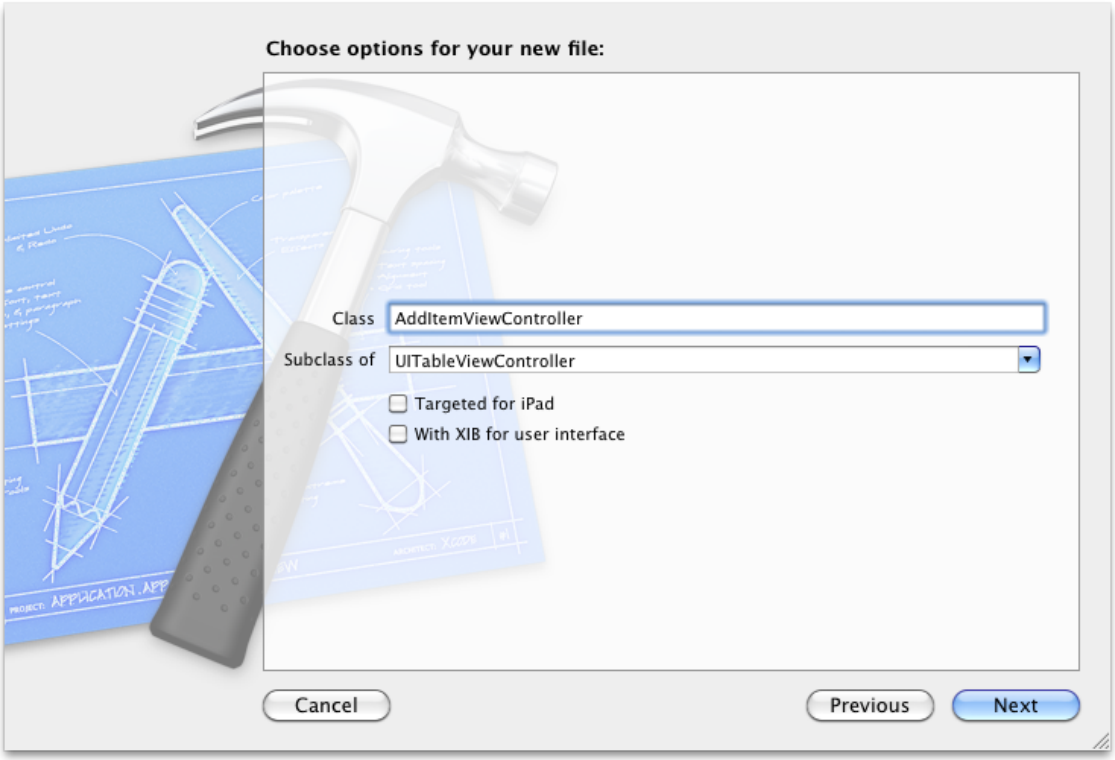
Choosing the Objective-C class template in Xcode 4.3



» In the next step, choose the following options:

- Class: `AddItemViewController`
- Subclass of: `UITableViewController` (you can pick that from the list)
- Targeted for iPad: Uncheck this
- With XIB for user interface: Uncheck this

Choosing the options for the UIViewController subclass



Note: Make sure the "Subclass of" field is set to "UI**Table**ViewController", not just "UIViewController"!

This adds two files to the project, AddItemViewController.h and AddItemViewController.m. There is no .xib file because we're using the Storyboard editor to build this screen, so the view controller doesn't need its own nib.

» Change AddItemViewController.h to add the two action methods:

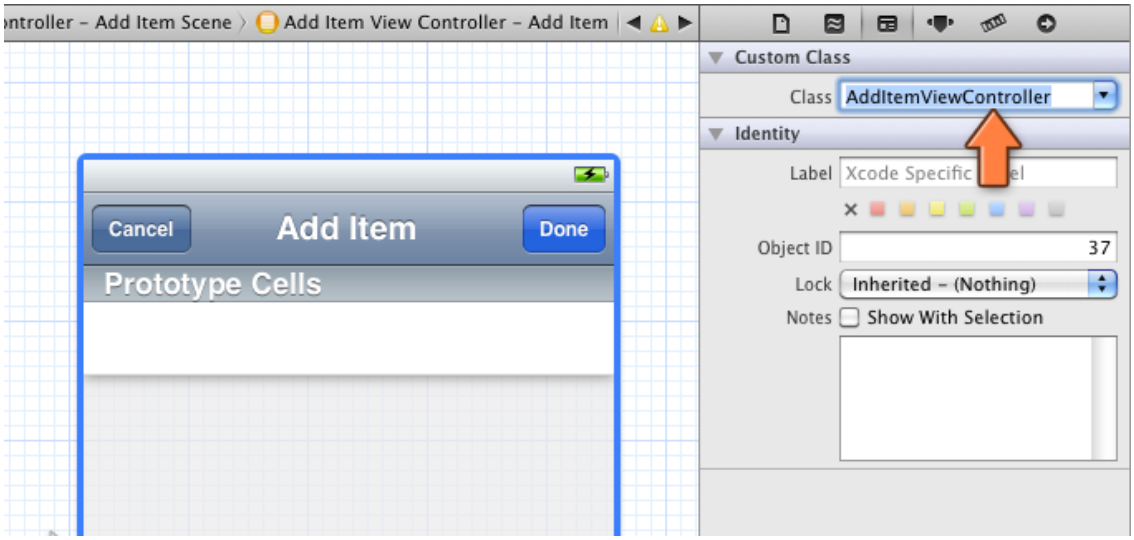
AddItemViewController.h

```
#import <UIKit/UIKit.h>

@interface AddItemViewController : UITableViewController

- (IBAction)cancel;
- (IBAction)done;

@end
```

» In the Storyboard editor, select the table view controller and go to the Identity Inspector. Under Custom Class, type "AddItemViewController".

This tells the Storyboard that the view controller from this scene is actually our `AddItemViewController` object.

Changing the class name of the AddItemViewController



Don't forget this step! Without it, the Add Item screen will simply not work. Also make sure that it is really the view controller that is selected before you change the fields in the Identity Inspector (the scene needs to have a fat blue border). A common mistake is to select the table view and change that.

With the Class field set, you can hook up the Cancel bar button to the `cancel` action and the Done bar button to the `done` action.

» Ctrl-drag from the bar buttons to anywhere else in that view controller's scene (for example, the status bar) and pick the proper action from the popup menu.

The final step is to implement the action methods in AddItemViewController.m. However, the Xcode template put a lot of stuff in this file that we don't need. The template assumes you'll fill this in before you run the app again. If you don't and try to run the app right now, Xcode will give many warnings. So let's get rid of that placeholder code first.

» In AddItemViewController.m, delete everything from the following line until `@end` (but not the `@end` line itself!):

```
#pragma mark - Table view data source
```

The lines you just deleted were the `numberOfRowsInSection`, `cellForRowAtIndexPath` and `didSelectRowAtIndexPath` methods that we've seen before plus a few other data

source and delegate methods for the table view. We won't need them for this particular view controller.

IMPORTANT! Do not skip this step. If you do not remove these methods, then the Add Item screen will not work properly.

» Add the `cancel` and `done` actions at the bottom of AddItemViewController.m, as always before `@end`:

AddItemViewController.m

```
- (IBAction)cancel
{
  [self.presentingViewController dismissViewControllerAnimated:YES completion:↵
      nil];
}

- (IBAction)done
{
  [self.presentingViewController dismissViewControllerAnimated:YES completion:↵
      nil];
}
```

This tells the "presenting view controller", which is the view controller that presented this modal screen, to close the screen with an animation. If you're wondering which of our four view controllers is the presenting one, it's the `UINavigationController` that contains the `ChecklistsViewController`, i.e. the one on the far left in the Storyboard.

What do you think happens to the `AddItemViewController` object when we dismiss it? After the view controller disappears from the screen, its object is destroyed and the memory it was using is reclaimed by the system. Every time the user opens the Add Item screen, we make a new instance for it. This means a view controller object is only alive for the duration that the user is interacting with it; there is no point in keeping it around afterwards.

Container view controllers

I've been saying that one view controller represents one screen, but here we actually have two view controllers for each screen. The app's main screen consists of the `ChecklistsViewController` inside a navigation controller, and the Add Item screen is composed of the `AddItemViewController` that sits inside its own navigation controller.

The Navigation Controller is a special type of view controller that acts as a container for other view controllers. It comes with a navigation bar and has the ability to easily go from one screen to another. The container essentially "wraps around" these screens. It's just the frame that contains the view controllers that do the real work, which are known as the "content" controllers.

Another often-used container is the Tab Bar Controller, which you'll see in the next tutorial. On the iPad, container view controllers are even more commonplace. View controllers on the iPhone are fullscreen but on the iPad they often occupy only a portion of the screen, such as the content of a popover or one of the panes in a split-view.

Static table cells

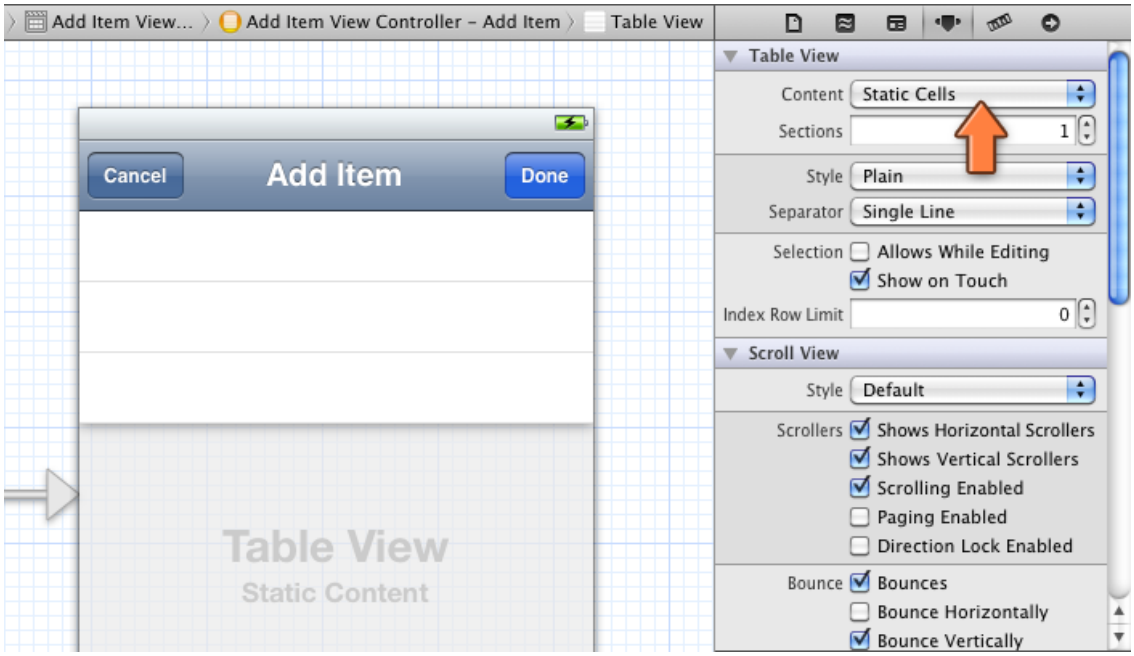
Let's change the look of the Add Item screen. Currently it is an empty table with a navigation bar on top, but I want it to look like this:

What the Add Item screen will look like when we're done

» Go to the Storyboard editor and select the Table View object inside the `AddItemViewController`. In the Attributes Inspector, change the Content setting from Dynamic Prototypes to Static Cells.

(Note that Xcode has been giving us a warning ever since we added this second table view controller: "Prototype table cells must have reuse identifiers." Switching to static cells gets rid of this warning.)

Changing to table view to static cells

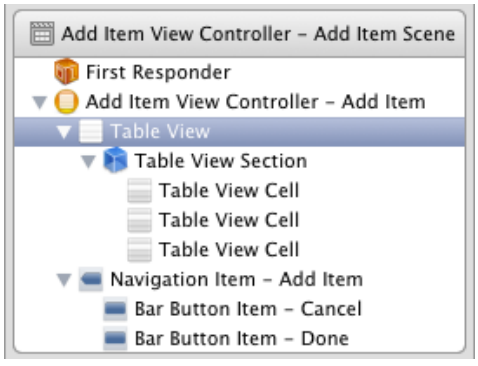


You use static cells when you know beforehand how many sections and rows the table view will have. This is handy for screens that require the user to enter data, such as the one we're building. We can design the rows directly in the Storyboard editor. For a table with static cells we don't need to provide a data source, and we can hook up the controls from the cells directly to properties on the view controller.

As you can see in the Scene pane on the left, the table view now has a Table View Section object hanging under it, and three Table View Cells in that section.

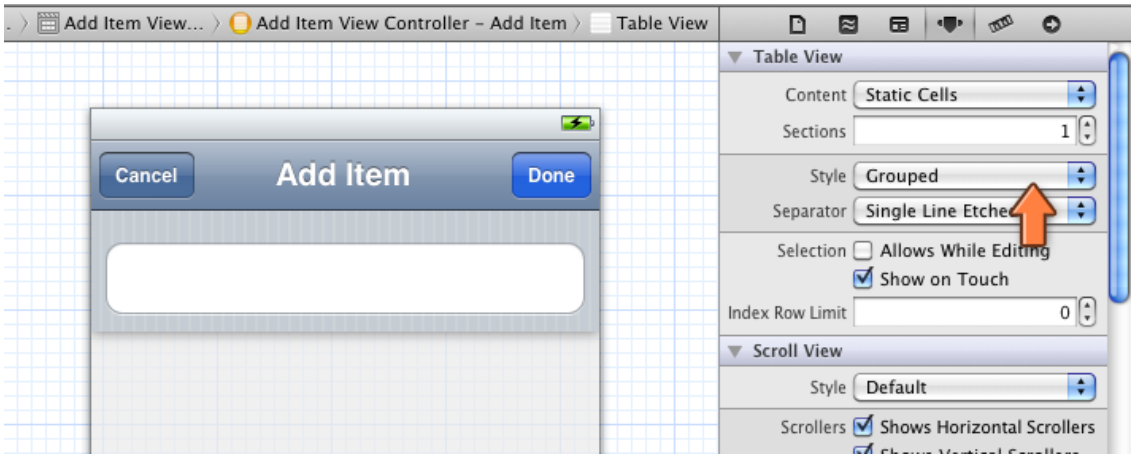
» Click on the bottom two cells and delete them. We only need one cell for now.

The table view has a section with three static cells



» Click the Table View again and in the Attributes Inspector set its Style to Grouped. That gives us the look we want.

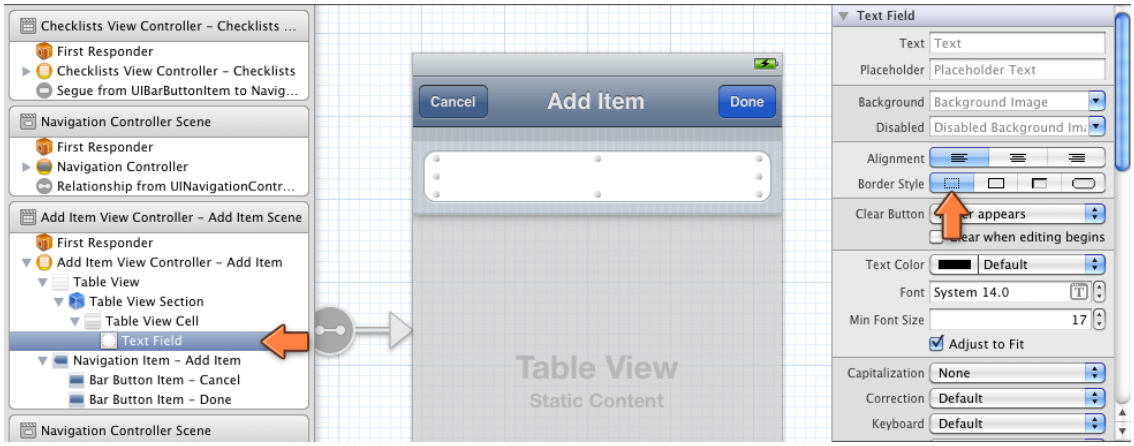
The table view with grouped style



Inside the table view cell we'll add a text field component that lets us type the text.

» Drag a Text Field into the cell and size it up nicely. In the Attributes Inspector for the text field, set the Border Style to none:

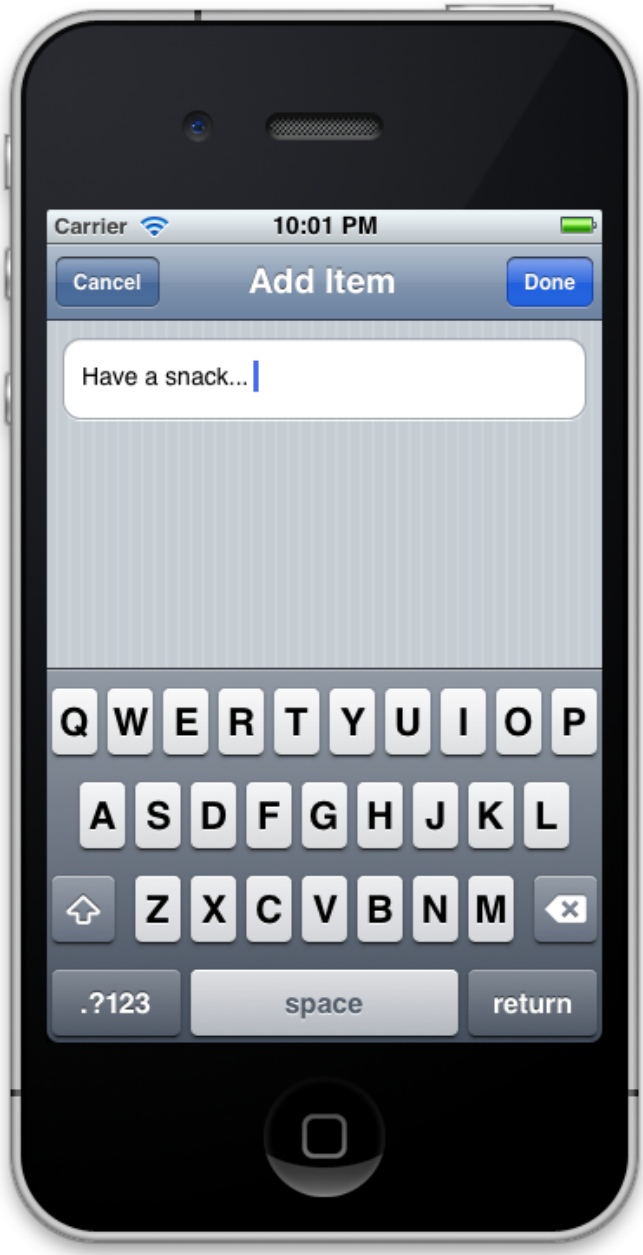
Adding a text field to the table view cell



» Run the app and press the + button to open the Add Item screen. Tap on the cell and you'll see the keyboard slide in from the bottom of the screen.

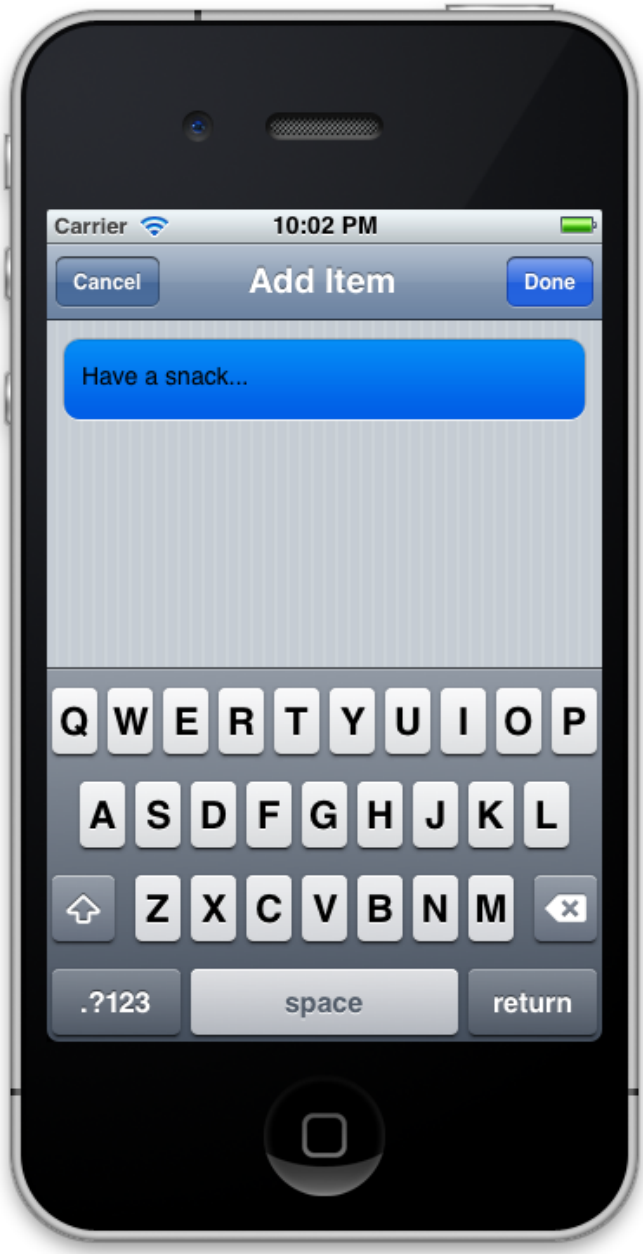
Any time you make a text field active, the keyboard automatically appears. You can type into the text field by tapping on the letters. (On the Simulator, you can simply type using your Mac's keyboard.)

You can now type text into the table view cell



But look what happens when you tap just outside the text field's area, but still in the cell:

Whoops, that looks a little weird



The row turns blue because you selected it. That's not what we want, so we should disable selections for this row.

» In the AddItemViewController.m file, add the following method at the bottom:

AddItemViewController.m

```
- (NSIndexPath *)tableView:(UITableView *)tableView willSelectRowAtIndexPath:(↵
    NSIndexPath *)indexPath
```

```
{
  return nil;
}
```

This is another table view delegate method. When the user taps in a row, the table sends the delegate a `willSelectRowAtIndexPath` message that says: “Hi delegate, I am about to select this particular row.” By returning `nil`, our delegate answers: “Sorry, but you’re not allowed to!”

There is one more thing we need to do to prevent the row from going blue. It’s already impossible to select the row, as we’ve just told the table view we won’t allow it. However, the cell also has a Selection Color property. This is set to Blue by default. Even if you make it impossible for the row to be selected, sometimes UIKit still briefly draws it blue when you tap it. Therefore it is best to also disable this selection color.

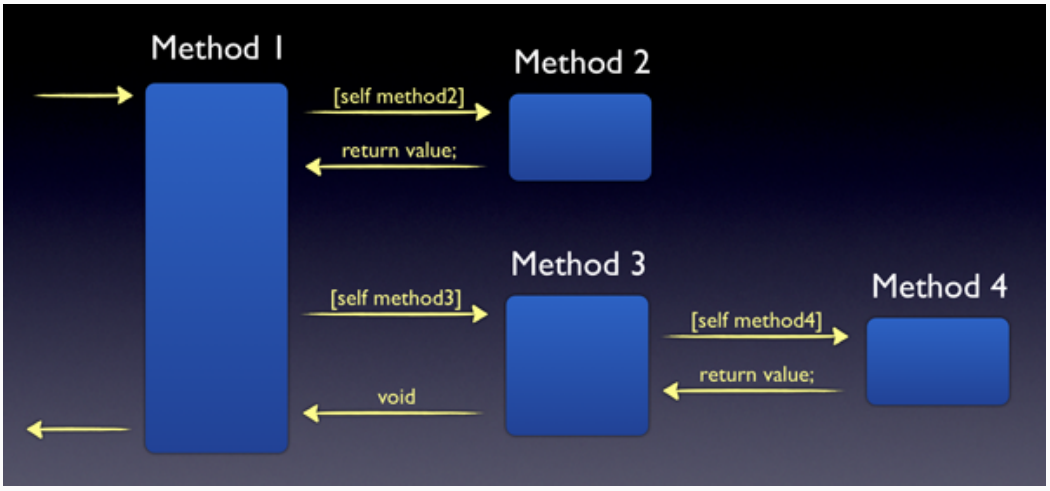
» In the Storyboard editor, select the table view cell and go to the Attributes Inspector. Set the Selection attribute to None.

Now if you run the app, it is impossible to select the row and make it turn blue.

Return to sender

We've seen the `return` statement a few times now. You use `return` to send a value from a method back to the method that called it. Let's take a more detailed look at what it does.

Methods call other methods and receive values in return



You cannot just return any value. The value you return must be of the datatype that is specified in front of the method name. For example, `shouldAutorotateToInterfaceOrientation` must return a `BOOL` value as its name begins with `(BOOL)`:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
  (UIInterfaceOrientation)interfaceOrientation
{
  return YES;
}
```

The value `YES` is a `BOOL` so that works. If instead you were to write:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
  (UIInterfaceOrientation)interfaceOrientation
{
  return @"Yes";
}
```

then the compiler would give an error message as `@"Yes"` is a string, not a `BOOL`. To a human reader they look similar and you'd easily understand the intent, but Objective-C isn't that tolerant. Datatypes have to match or it just isn't allowed.

The `numberOfRowsInSection` method is supposed to return an `NSInteger`, which is another name for `int`, so the statement "return 1" is perfectly valid:

```
- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section
{
```

```
  return 1;

}
```

Our most recent version of this method looks like this:

```
- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section
{
  return [items count];
}
```

That is also a valid return statement because the `count` method from `NSArray` returns an `NSUInteger` value. The differences between `NSInteger`, `NSUInteger` and `int` are not very interesting. The important thing is that Objective-C can easily convert between them as they all represent whole numbers (integers).

The `cellForRowAtIndexPath` method is supposed to return a `UITableViewCell` object:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView
    dequeueReusableCellWithIdentifier:@"ChecklistItem"];

  . . .

  return cell;
}
```

The difference with the previous examples is that `BOOL`, `int` and `NSInteger` are not objects but `UITableViewCell` is. You can tell the difference by the `*`. A name followed by an asterisk means that we're dealing with an object. The others are so-called *primitive types*. The next tutorial explains more about the differences between objects and primitive types. For now, just keep an eye out for the `*`.

The `willSelectRowAtIndexPath` method is supposed to return an `NSIndexPath` object. However, we can also make it return "nil", which means no object.

```
- (NSIndexPath *)tableView:(UITableView *)tableView
  willSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
  return nil;
}
```

When a method is expected to return an object, you can return `nil` as well. That doesn't mean `nil` is always a proper response. For example, if we were to return `nil` from `cellForRowAtIndexPath`, the app would crash. Some methods don't like it when you return `nil` instead of an actual object, but for others it is OK. How do you know which is the case? You can find that in the documentation of the method in question.

In the case of `willSelectRowAtIndexPath`, the iOS documentation [http://developer.apple.com/library/ios/DOCUMENTATION/UIKit/Reference/UITableViewDelegate_Protocol/Reference/Reference.html#//apple_ref/occ/intfm/UITableViewDelegate/tableView:willSelectRowAtIndexPath:] says:

> "Return Value: An index-path object that confirms or alters the selected row. Return an NSIndexPath object other than indexPath if you want another cell to be selected. Return nil if you don't want the row selected."

This means we can either:

1. Return the same index-path we were given. This confirms that this row can be selected.
2. Return another index-path in order to select another row.
3. Return `nil` to prevent the row from being selected, which is what we did.

The documentation [http://developer.apple.com/library/ios/documentation/uikit/reference/UITableViewDataSource_Protocol/Reference/Reference.html#//apple_ref/occ/intfm/UITableViewDataSource/tableView:cellForRowAtIndexPath:] for `cellForRowAtIndexPath` says:

> "Return Value: An object inheriting from UITableViewCell that the table view can use for the specified row. An assertion is raised if you return nil."

In other words, this method must always return a proper table view cell object.

"An assertion is raised if you return nil" means: returning `nil` instead of a valid `UITableViewCell` object will crash the app on purpose because you're doing something you're not supposed to. An *assertion* is a special debugging tool that is used to check that your code always does something valid. If not, the app will crash with a helpful error message. We'll see more of this later when we talk about finding bugs — and squashing them.

You've also seen methods that do not return anything:

```
- (void)viewDidLoad
```

and

```
- (IBAction)cancel
```

The term "void" means: this method does not pass a value back to the caller. `IBAction` is a synonym for `void` but as you know it's also a special symbol that lets Interface Builder and the Storyboard editor know that this method can be hooked up to a button or other control. Because `IBAction` means the same as `void`, action methods never return a value.

If you forget to return a value from a method that expects to return something, then Xcode will give you the following warning: "Control reaches end of non-void function". The app will still run, but there's a big chance it will crash at some point, as whoever calls that method is expecting a to receive a value but the method is not returning any. As always, pay attention to the warnings of Xcode!

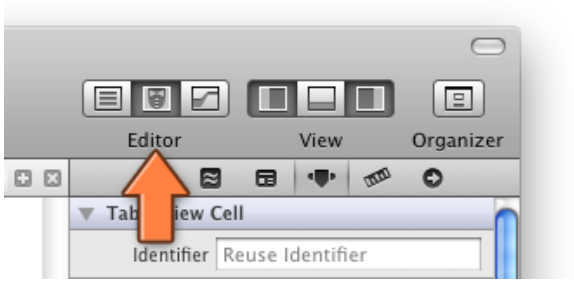
Reading the text from the text field

We now have a text field in our table view cell that the user can type into, but how do we read the text that the user has typed? When the user taps Done, we need to get that text and somehow put it into a new `ChecklistItem` and add it to our list of items. This means the `done` action needs to be able to refer to the text field. You already know how to refer to controls from within your view controller: use a property.

When we added outlets in the previous tutorial, I told you to type in the `@property` declaration and add a `@synthesize` line. I'm going to show you a trick now that will save you some typing. You can let the Storyboard editor do all of this automatically by Ctrl-dragging from the control in question into your .h file.

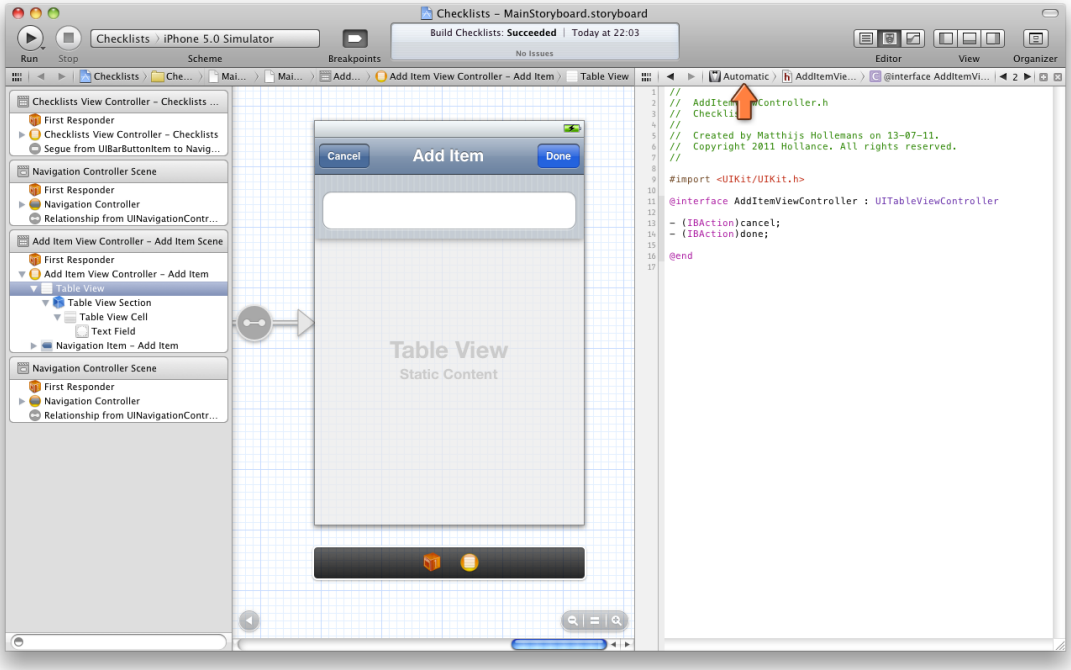
» First, open the Assistant editor, using the toolbar button:

Click the toolbar button to open the Assistant editor



This may make the screen a little crowded — there are now five horizontal panels open — so you might want to close the Project Navigator and the Utilities pane using the "View" toolbar buttons.

The Assistant editor opens a new pane on the right of the screen. In the Jump Bar it should say "Automatic":

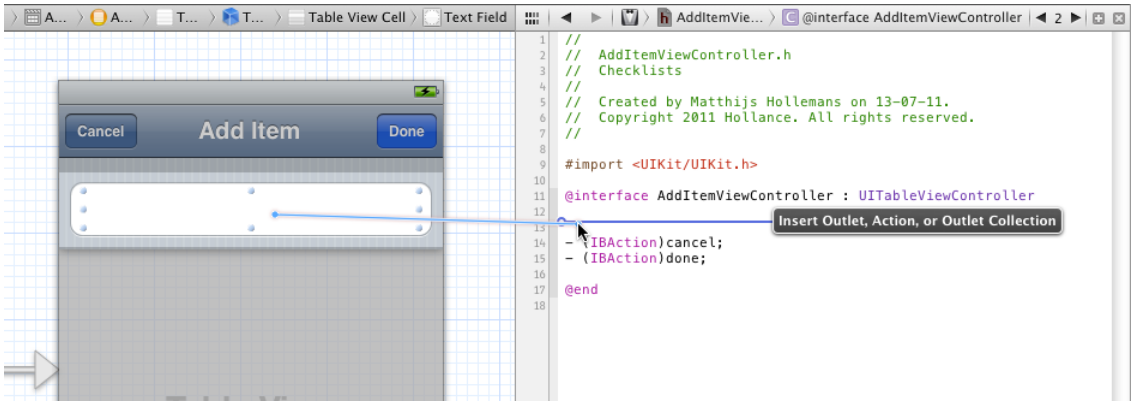


The Assistant editor should have automatically chosen the AddItemViewController.h file for you. “Automatic” means the Assistant editor figures out what other file is related to the one you’re currently editing. If you’re editing the Storyboard, the related file is the selected view controller’s .h file.

(Sometimes Xcode can be a little dodgy here. If it shows you something else than AddItemViewController.h, try de-selecting and re-selecting the view controller, or click around in the Jump Bar a bit. Switching to another file and then back to the Storyboard may also help.)

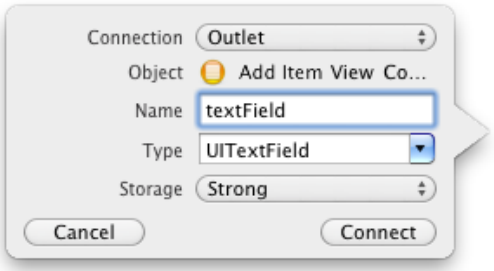
» With the view controller and the .h file side by side, select the text field. Then Ctrl-drag from the text field into the .h file:

Ctrl-drag from the text field into the .h file



When you let go, a popup appears:

The popup that lets you add a new outlet



» Choose the following options:

- Connection: Outlet
- Name: textField
- Type: `UITextField`
- Storage: Strong

» Press Connect and voila, Xcode has automatically inserted a `@property` for us.

The AddItemViewController.h file now looks like this:

AddItemViewController.h

```
#import <UIKit/UIKit.h>

@interface AddItemViewController : UITableViewController

@property (strong, nonatomic) IBOutlet UITextField *textField;

- (IBAction)cancel;
```

```
- (IBAction)done;

@end
```

In the AddItemViewController.m file, a synthesize statement was added:

AddItemViewController.m

```
@synthesize textField;
```

Very handy that Xcode now does this for us. As of Xcode 4.4, it is no longer necessary to type `@synthesize` for your properties, so if you're using the latest version of Xcode, nothing was actually added to your AddItemViewController.m. Fine with me… the less code we need, the better!

We have now hooked up the text field object with a new property named `textField`. Now we'll modify the `done` action to write the contents of this text field to the Debug Area. This is a quick way to verify that we can actually read what the user typed.

» Change the `done` action to:

AddItemViewController.m

```
- (IBAction)done
{
  NSLog(@"Contents of the text field: %@", self.textField.text);

  [self.presentingViewController dismissViewControllerAnimated:YES completion:↵
      nil];
}
```

» Run the app, press the + button and type something in the text field. When you press Done, the Add Item screen should close and Xcode should open the Debug pane with a message like this:

```
Checklists[1165:207] Contents of the text field: Hello, world!
```

Great, so that works. `NSLog()` should be an old friend by now. You've seen the `%d` and `%f` format specifiers before, which were used for integer values and floating-point values (decimals), respectively. The `%@` specifier is used to print out the value of an object, in this case the contents of the text field's `text` property.

Polishing it up

Before we will write the code to take this text and insert it as a new item into the list, let's improve the design and workings of the Add Item screen a little. For instance, it would be nice if you didn't have to tap into the text field in order to bring up the keyboard. It would be more convenient if the keyboard automatically appeared once the screen opens.

» To accomplish this, add the following line to the `viewWillAppear` method:

AddItemViewController.m

```
- (void)viewWillAppear:(BOOL)animated
{
  [super viewWillAppear:animated];

  [self.textField becomeFirstResponder];
}
```

Xcode 4.2 already put a basic version of this method in our view controller as it is commonly needed by view controllers. However, the template for Xcode 4.3 does not include it, so simply copy-paste the whole thing into the .m file.

The view controller receives the `viewWillAppear` message just before it becomes visible. That is a perfect time to make the text field active. We do this by sending it the `becomeFirstResponder` message. If you've done programming on other platforms, this is often called "giving the control focus". In iOS terminology, the control becomes the *first responder*.

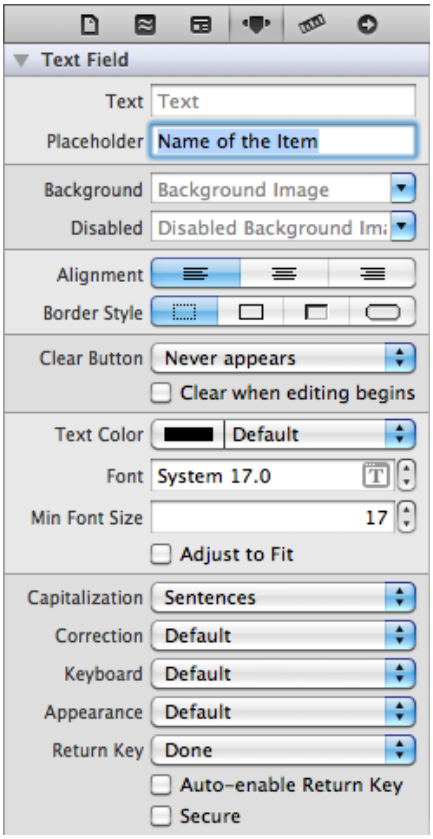
If you now run the app and go to the Add Item screen, you can start typing right away.

Let's style the input field a bit.

» Go to the Storyboard editor and select the text field. Go to its Attributes Inspector and set the following attributes:

- Placeholder: Name of the Item
- Font: System 17
- Adjust to Fit: Uncheck this
- Capitalization: Sentences
- Return Key: Done

The text field attributes



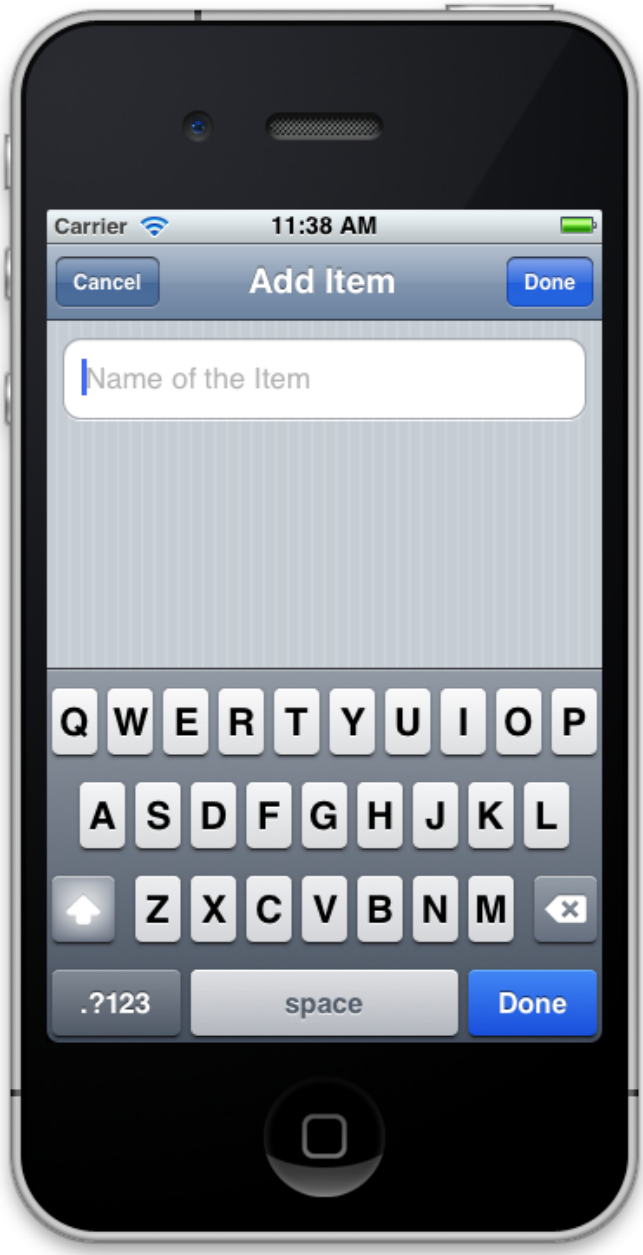
There are several options here that let you configure the keyboard that appears when the text field becomes active. If this were a field that only allowed numbers, for example, you would set the Keyboard to Number Pad. If it were an email address field, you'd set it to E-mail Address. For our purposes, the Default keyboard is appropriate.

You can also change the text that is displayed on the keyboards's Return Key. By default it says "return" but we set it to "Done". This is just the text on the button. We'll make the keyboard's Done button trigger the same action as the Done button from the navigation bar.

» Make sure the text field is selected and open the Connections Inspector. Drag from the Did End on Exit event to the view controller and pick the `done` action.

When you run the app now, pressing Done on the keyboard will also close the screen and print the text to the Debug Area.

The keyboard now has a big blue Done button



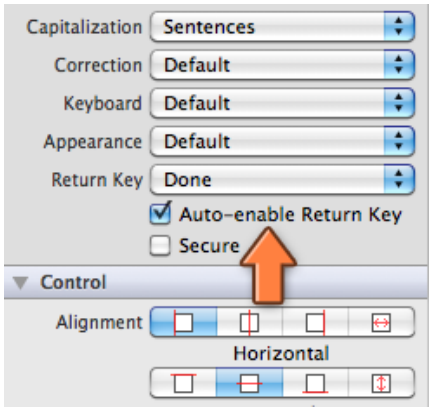
It's always good to validate the input from the user to make sure what they're entering is acceptable. For instance, what should happen if the user immediately taps the Done button on the Add Item screen without entering any text? Adding a to-do item to the list that has no text is not very useful, so in order to prevent this we should disable the Done button when no text has been typed yet.

We have two Done buttons, one on the keyboard and one in the navigation bar. Let's start with the Done button from the keyboard as this is the simplest one to fix.

» On the Attributes Inspector for the text field, check Auto-enable Return Key.

That's it. Now when you run the app the Done button on the keyboard automatically is disabled when there is no text in the text field. Try it out!

The Auto-enable Return Key option disables the return key when there is no text



For the Done button in the navigation bar we have to do a little more work. We have to check the contents of the text field after every keystroke to see if it is now empty or not. If it is, then we disable the button. The user can always press Cancel but Done only works when there is text.

In order to listen to changes to the text field — which may come from taps on the keyboard but also from cut/paste — we need to make the view controller a delegate for the text field. The text field will send events to this delegate to let it know what is going on. The delegate, which will be our `AddItemViewController`, can then respond to these events and take appropriate actions.

We've seen delegates before with `UITableView`. The text field object, `UITextField`, also has a delegate. These are two different delegates and we make the view controller play both roles. Later in this tutorial we'll add even more delegates.

» Add `UITextFieldDelegate` to AddItemViewController.h's `@interface` line:

AddItemViewController.h

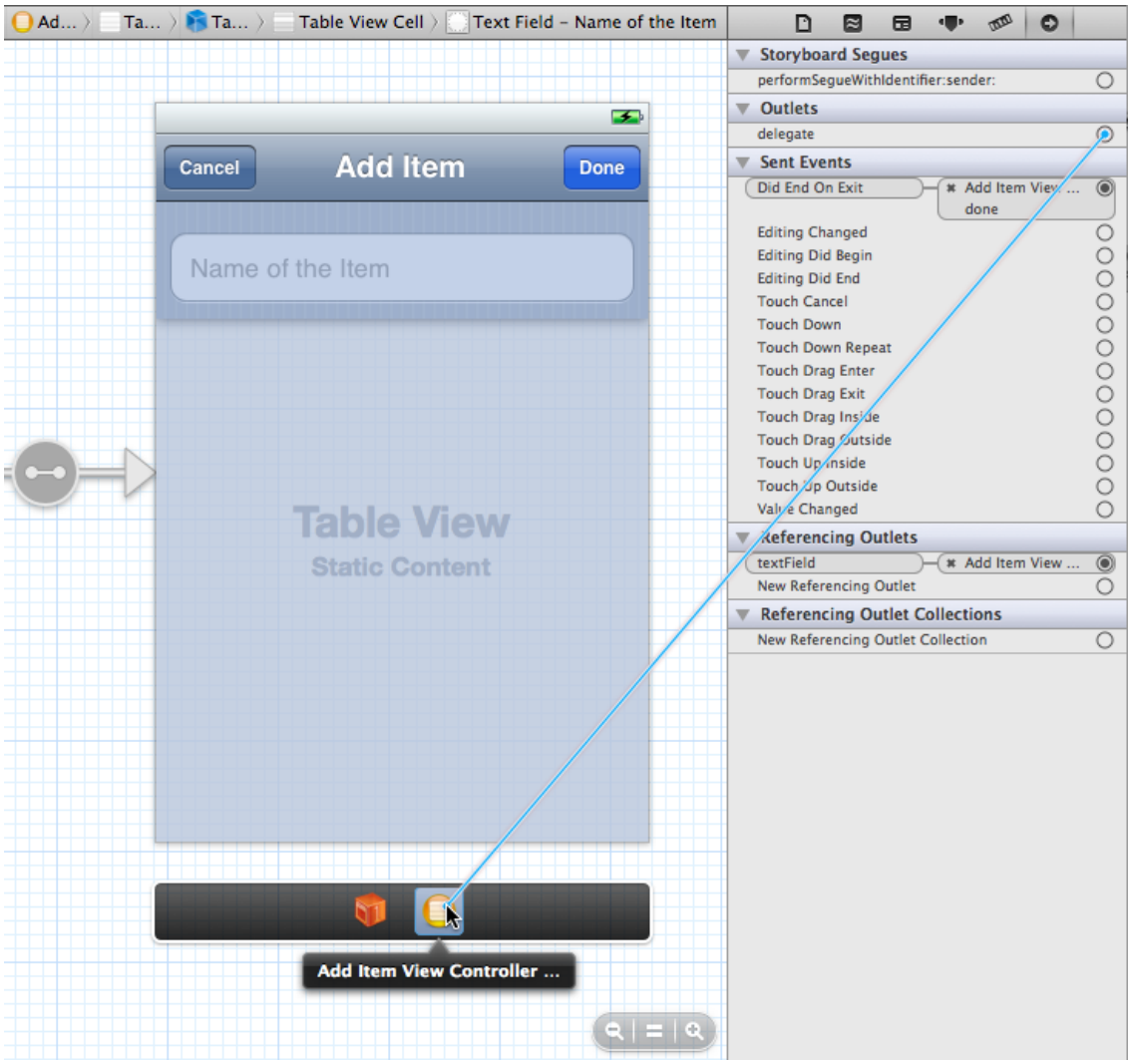
```
@interface AddItemViewController : UITableViewController <UITextFieldDelegate>
  . . .
```

The view controller now says: I can be a delegate for text field objects.

We also have to tell the text field that we have a delegate for it.

» Go to the Storyboard editor and select the text field. There are several different ways in which you can hook up the text field's delegate outlet to the view controller. I prefer to go to its Connections Inspector and drag from "delegate" to the view controller's icon in the dock:

Drag from the Connections Inspector to connect the text field delegate



We also have to add a property for the Done button, so we can send it messages from within the view controller in order to enable or disable it.

» Open the Assistant editor and make sure AddItemViewController.h is visible in the newly opened pane. Ctrl-drag from the Done bar button to AddItemViewController.h and let go. Name the new outlet `doneBarButton`.

This adds the following property:

AddItemViewController.h

```
@property (nonatomic, strong) IBOutlet UIBarButtonItem *doneBarButton;
```

Xcode also automatically added the corresponding `@synthesize` statement for this property in AddItemViewController.m. (Not with Xcode 4.4 or up, where it is no longer required.)

» Add the following to AddItemViewController.m, at the bottom:

AddItemViewController.m

```
- (BOOL)textField:(UITextField *)theTextField shouldChangeCharactersInRange:(↵
    NSRange)range replacementString:(NSString *)string
{
  NSString *newText = [theTextField.text stringByReplacingCharactersInRange:↵
      range withString:string];
  if ([newText length] > 0) {
    self.doneBarButton.enabled = YES;
  } else {
    self.doneBarButton.enabled = NO;
  }
  return YES;
}
```

This is one of the `UITextField` delegate methods. It is invoked every time the user changes text, whether by tapping on the keyboard or by cut/paste.

First, we figure out what the new text will be:

```
NSString *newText = [theTextField.text stringByReplacingCharactersInRange:range↵
     withString:string];
```

The `shouldChangeCharactersInRange` delegate method doesn't give us the new text, only which part of the text should be replaced (the range) and the text it should be replaced with (the replacement string). So we calculate ourselves what the new text will be by taking the text field's text and doing the replacement. This gives us a new string object that we store in the `newText` local variable.

Then we check if the new text is empty by looking at its length, and enable or disable the Done button accordingly:

```
if ([newText length] > 0) {
  self.doneBarButton.enabled = YES;
} else {
  self.doneBarButton.enabled = NO;
}
```

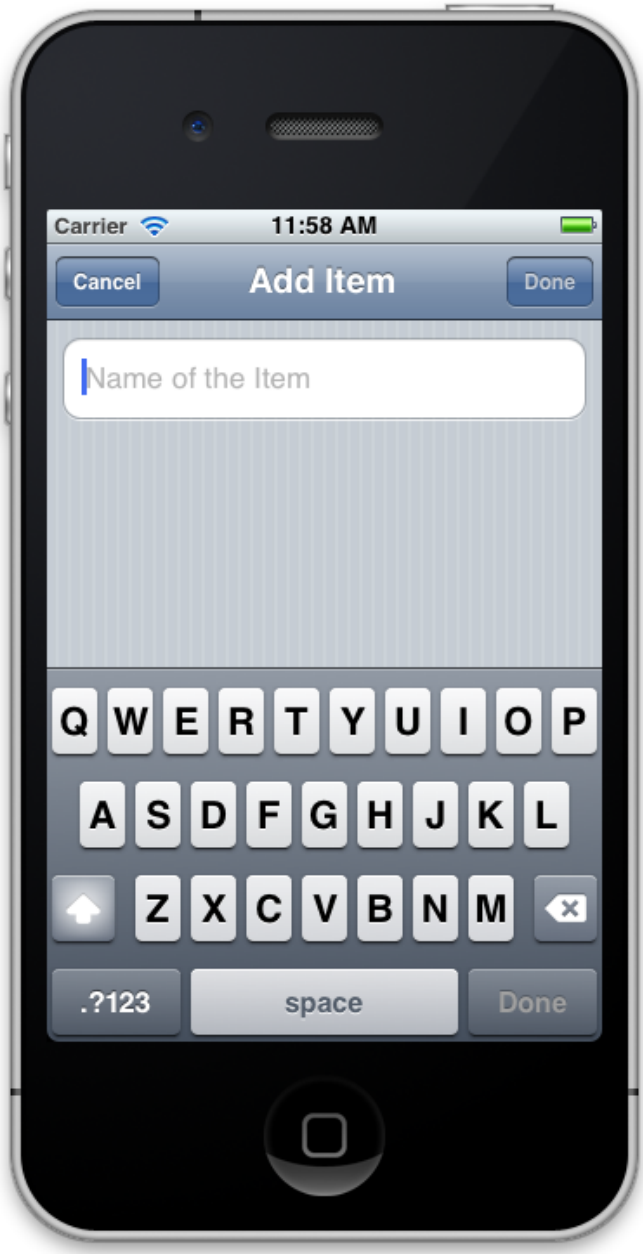
» Run the app and type some text into the text field. Now remove that text and you'll see that the Done button in the navigation bar properly gets disabled when the text field becomes empty.

One problem: The Done button is initially enabled when the Add Item screen opens, but there is no text in the text field at that point so it really should be disabled. This is simple enough to fix:

» In the Storyboard editor, select the Done bar button and go to the Attributes Inspector. Uncheck the Enabled box.

The Done buttons are now properly disabled when there is no text in the text field:

You cannot press Done if there is no text



There is actually a slightly better way to write the above method:

AddItemViewController.m

```
- (BOOL)textField:(UITextField *)theTextField shouldChangeCharactersInRange:(↵
    NSRange)range replacementString:(NSString *)string
{
  NSString *newText = [theTextField.text stringByReplacingCharactersInRange:↵
      range withString:string];
```

```
  self.doneBarButton.enabled = ([newText length] > 0);
  return YES;
}
```

We replaced the if-statement by the line:

```
  self.doneBarButton.enabled = ([newText length] > 0);
```

Earlier we did:

```
  if ([newText length] > 0) {
    // we get here if the length is greater than 0
  } else {
    // we get here if the length is 0
  }
```

We check the condition `[newText length] > 0`. If that condition is true, i.e. the text length is greater than 0, we set `doneBarButton`'s `enabled` property to `YES`. If the condition is false, we set the `enabled` property to `NO`.

Notice that in these sentences I'm basically saying: if the condition is `YES` then `enabled` becomes `YES` but if the condition is `NO` then `enabled` becomes `NO`. In other words, we always set the `enabled` property to the result of the condition: `YES` or `NO`.

That makes it possible for us to skip the if, and simply do:

```
  self.doneBarButton.enabled = the result of the condition;
```

which in Objective-C reads as follows:

```
  self.doneBarButton.enabled = ([newText length] > 0);
```

The ( ) parentheses are not really necessary, you can also write it like this:

```
  self.doneBarButton.enabled = [newText length] > 0;
```

However, I find this to be slightly less readable, so I use the parentheses to make it clear beyond a doubt that `[newText length] > 0` is evaluated first and that the assignment takes place after that.

If `[newText length]` is greater than 0, `self.doneBarButton.enabled` becomes `YES`; otherwise it becomes `NO`.

We can fit this into a single statement because the *relational operators* all return `YES` if the condition is true and `NO` if the condition is false. These are the relational operators in Objective-C:

> (greater than)

< (less than)

>= (greater than or equal to)

<= (less than or equal to)

== (equal)

!= (not equal)

Remember this trick; whenever you see code like this,

```
  if (some condition) {
    something = YES;
  } else {
    something = NO;
  }
```

then you can write it simply as:

```
  something = (some condition);
```

We've actually done this before in the `shouldAutorotateToInterfaceOrientation` method. It currently looks like this:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)↵
    interfaceOrientation
{
  return (interfaceOrientation == UIInterfaceOrientationPortrait);
}
```

Which is equivalent to:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)↵
    interfaceOrientation
{
  if (interfaceOrientation == UIInterfaceOrientationPortrait) {
    return YES;
  } else {
    return NO;
  }
}
```

In practice it doesn't really matter which version you use. I prefer the shorter one; that's what the pros do. Just remember that relational operators such as == and > always return `YES` or `NO`, so the extra `if` really isn't necessary.

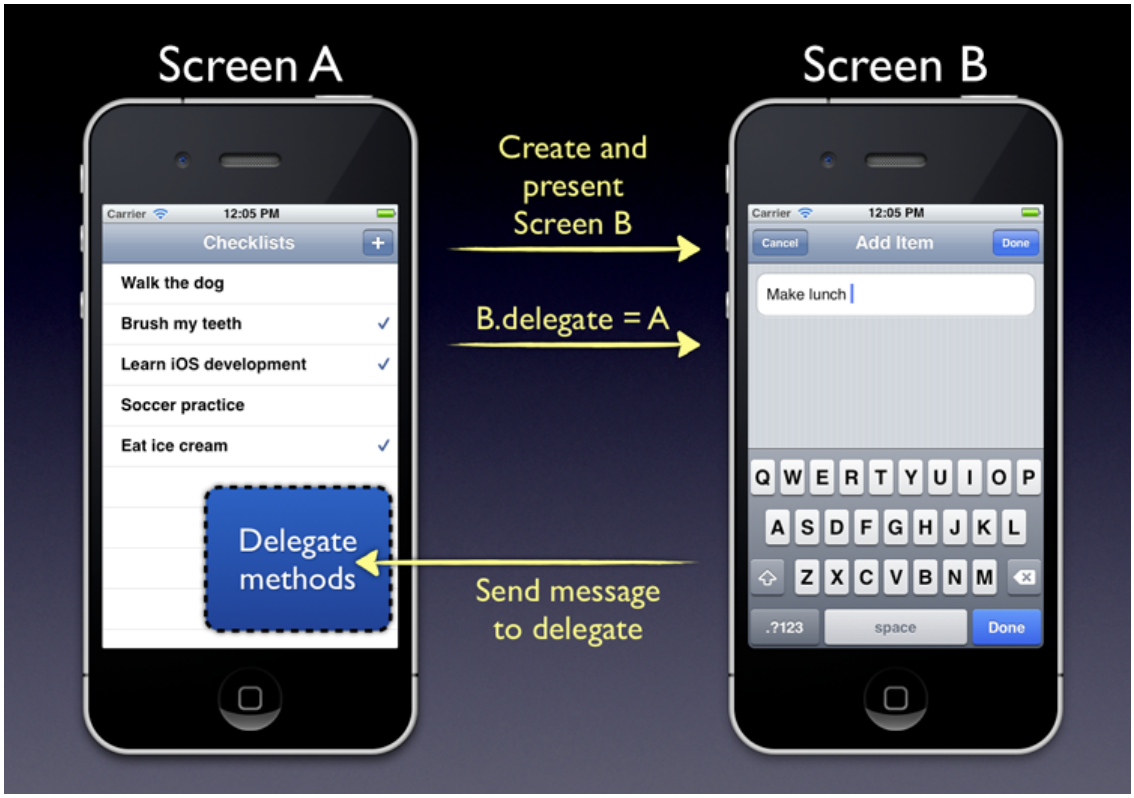
Adding new ChecklistItems

We now have an Add Item screen that lets the user enter text. We properly validate the input so that we'll never end up with text that is empty. But how do we get this text into a new `ChecklistItem` that we can add to the list? To do this we will have to make our own delegate.

You've already seen delegates in a few different places: The table view has a delegate that responds to taps on the rows. The text field has a delegate that we use to validate the length of the text. In the Bull's Eye tutorial we used a delegate to listen to the alert view. And our app also has something named the `ChecklistsAppDelegate` (see the Project Navigator). You can't turn a corner in this place without bumping into a delegate…

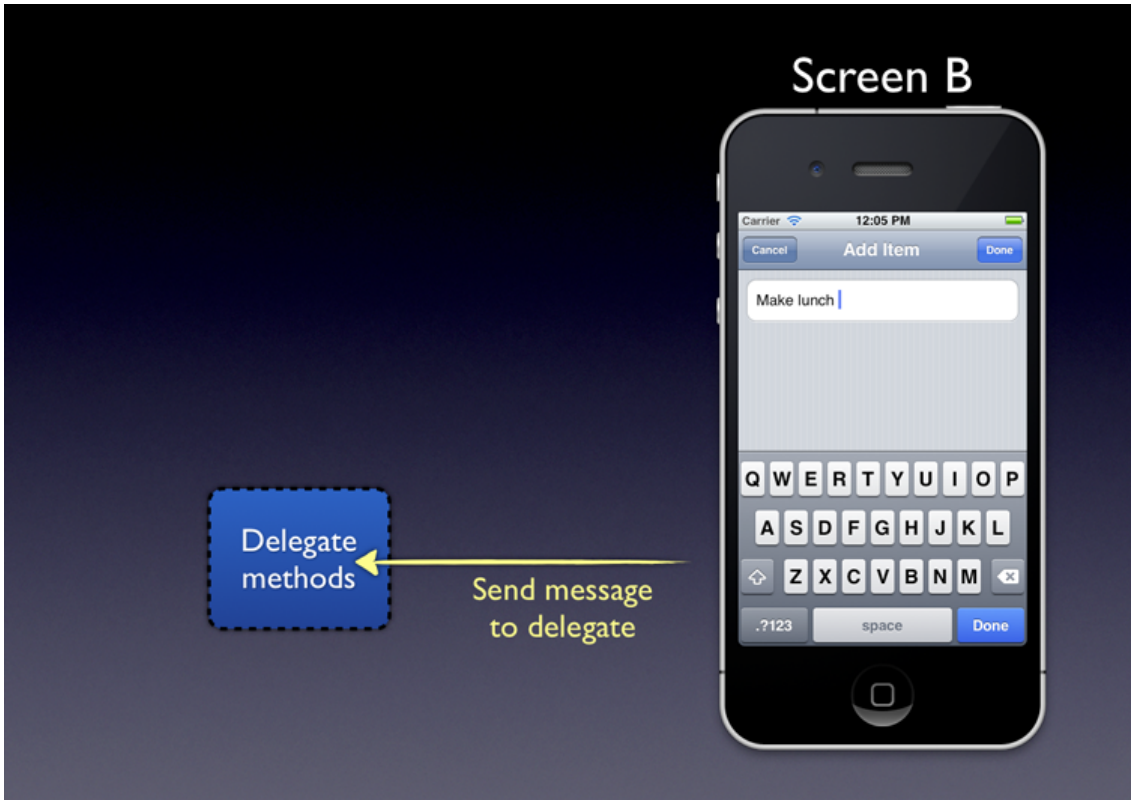
The delegate pattern is commonly used to handle the situation we find ourselves in: screen A opens screen B and at some point screen B needs to communicate back to screen A, for example when it closes. The solution is to make A a delegate of B so screen B can send its messages to A whenever it needs to.

Screen A launches screen B and becomes its delegate



The cool thing about the delegate pattern is that screen B doesn't really know anything about screen A. It just knows that *some* object is its delegate. Other than that, screen B doesn't care who that is. Just like `UITableView` doesn't really care about our view controller, only that it delivers table view cells when the table view asks for them. This principle, where screen B is independent of screen A, yet can still talk to it, is called *loose coupling* and is considered good object-oriented design practice.

This is what Screen B sees: only the delegate part, not the rest of screen A



We will use the delegate pattern to let the `AddItemViewController` send notifications to the `ChecklistsViewController`, without it having to know anything about this object.

» At the top of AddItemViewController.h, add this in between the `#import` and `@interface` lines:

AddItemViewController.h

```
@class AddItemViewController;
@class ChecklistItem;

@protocol AddItemViewControllerDelegate <NSObject>

- (void)addItemViewControllerDidCancel:(AddItemViewController *)controller;

- (void)addItemViewController:(AddItemViewController *)controller ↵
    didFinishAddingItem:(ChecklistItem *)item;

@end
```

This defines the `AddItemViewControllerDelegate` protocol. You should recognize the lines inside the `@protocol ... @end` block as method declarations.

Protocols

In Objective-C, a *protocol* doesn't have anything to do with computer networks or meeting royalty. It is simply a name for a group of methods. A protocol doesn't have instance variables and it doesn't implement any of the methods it declares. It just says: any object that conforms to this protocol must implement methods X, Y and Z.

The methods listed in the `AddItemViewControllerDelegate` protocol are `addItemViewControllerDidCancel:` and `addItemViewController:didFinishAddingItem:`. Delegates often have very long method names!

If we make the `ChecklistsViewController` conform to this protocol, then it must implement these two methods. The trick is that from then on we can refer to the `ChecklistsViewController` using the protocol name. That is done using the following syntax:

```
id <AddItemViewControllerDelegate> delegate;
```

The variable `delegate` is now a reference to some object that implements the methods of this protocol. You can send messages to the object in the `delegate` variable, without knowing what kind of object it really is. Of course, we know it is the `ChecklistsViewController` but `AddItemViewController` doesn't need to be aware of that. All it sees is `delegate`.

It is customary for the delegate methods to have a reference to their owner as the first (or only) parameter. That's why we pass along the `AddItemViewController` to both methods. This is not required but still a good idea. For example, it may happen that a table view delegate is the delegate for more than one table view. In that case, it needs to be able to distinguish between those two table views. That's why the table view delegate methods contain a reference to the `UITableView` object that sent the notification.

If you've programmed in other languages before, you may recognize protocols as being very similar to "interfaces". When the designers of the Objective-C language added protocols they couldn't use the term interface to describe protocols as this keyword was already used to declare objects (the `@interface` line in the .h files). Therefore `@protocol` it is. It's only a name, the concept is the same.

Notice the line that says `@class ChecklistItem;`. This tells the delegate protocol about the `ChecklistItem` object. In the past we've used `#import` to let one object know about other objects. So what's the difference between these two?

- The line `@class ChecklistItem;` simply says to the compiler: if you see the name `ChecklistItem` then that's an object we're going to be using. It doesn't tell the compiler exactly what that object does, just that it's an object. You'll use `@class` mostly in .h files because at that point the compiler doesn't need to know what the object's properties and methods are, just that it exists. This is also known as a *forward declaration*.
- `#import "ChecklistItem.h"` literally adds the contents of the ChecklistItem.h file into the current file when it is being compiled. More about this in the next tutorial,

but it means that when you use `#import`, the compiler knows everything about that object. If you want to access the properties of an object or call any of its methods, you need to use `#import`.

We need this forward declaration for the `ChecklistItem` object because we are using that object in the `AddItemViewControllerDelegate` protocol, as a parameter for the method `addItemViewController:didFinishAddingItem:`. Likewise for the forward declaration of `@class AddItemViewController`.

We're not done yet in AddItemViewController.h. The view controller must have a property that it can use to refer to the delegate.

» Add this inside the `@interface` block, below the other properties:

AddItemViewController.h

```
@property (nonatomic, weak) id <AddItemViewControllerDelegate> delegate;
```

» Because we're adding this property by hand, we also have to add a `@synthesize` line to AddItemViewController.m file, below the other `@synthesize` lines:

AddItemViewController.m

```
@synthesize delegate;
```

» Replace the `cancel` and `done` actions with the following:

AddItemViewController.m

```
- (IBAction)cancel
{
  [self.delegate addItemViewControllerDidCancel:self];
}

- (IBAction)done
{
  ChecklistItem *item = [[ChecklistItem alloc] init];
  item.text = self.textField.text;
  item.checked = NO;

  [self.delegate addItemViewController:self didFinishAddingItem:item];
}
```

Because `AddItemViewController` doesn't know yet what a `ChecklistItem` object is, we have to import its definition.

» Add the following line at the top of the file:

AddItemViewController.m

```
#import "ChecklistItem.h"
```

When the user taps the Cancel button, we send the `addItemViewControllerDidCancel` message to the delegate. We do something similar for the Done button, except that the message is `addItemViewController:didFinishAddingItem:` and we pass along a new `ChecklistItem` object.

If you were to run the app now, the Cancel and Done buttons would no longer appear to work. (Try it out!) That is because we haven't told the Add Item screen yet who its delegate is. That means `self.delegate` is `nil` and the messages aren't being sent to anyone; there is no one listening for them.

Tip: If you've programmed in other languages before, you may be afraid of the dreaded "null pointer dereference". In Objective-C it is perfectly valid to send a message to `nil`. This will not crash your app. You don't have to add an `if (self.delegate != nil)` check before sending a message to the delegate. The only thing you don't want to send messages to are deallocated objects (so-called zombies), but everyone knows you don't mess with zombies.

We will make the `ChecklistsViewController` suitable to be a delegate for `AddItemViewController`.

» Change ChecklistsViewController.h to:

ChecklistsViewController.h

```
#import <UIKit/UIKit.h>
#import "AddItemViewController.h"

@interface ChecklistsViewController : UITableViewController <↵
    AddItemViewControllerDelegate>

- (IBAction)addItem;

@end
```

The `#import` line for AddItemViewController.h is necessary to load the definition of the `AddItemViewControllerDelegate`, otherwise we cannot use it.

The change to the `@interface` line tells the compiler that `ChecklistsViewController` now conforms to the `AddItemViewControllerDelegate` protocol. Anytime you see some-

thing in between < > brackets on an `@interface` line, that means the object implements a particular protocol.

» Add the implementations of the protocol's methods at the bottom of ChecklistsViewController.m:

ChecklistsViewController.m

```
- (void)addItemViewControllerDidCancel:(AddItemViewController *)controller
{
  [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)addItemViewController:(AddItemViewController *)controller ↵
    didFinishAddingItem:(ChecklistItem *)item
{
  [self dismissViewControllerAnimated:YES completion:nil];
}
```

Currently these methods simply close the Add Item screen. Once we have this working we'll add the code to make the new `ChecklistItem` and add it to the table. Note that this is what the `AddItemViewController` used to do itself in its `cancel` and `done` actions. We've simply moved that responsibility to the delegate now.

In review, these are the steps for setting up the delegate pattern between two objects, where object A is the delegate for object B and object B will send out the messages:

1. Define a delegate `@protocol` for object B.
2. Give object B a property for that delegate protocol.
3. Make object B send messages to its delegate when something interesting happens, such as the user pressing the Cancel or Done buttons, or when it needs a piece of information.
4. Make object A conform to the delegate protocol. It should put the name of the protocol in its `@interface` line and implement the methods from the protocol.
5. Tell object B that object A is now its delegate.

This means there is one more thing we need to do: we have to tell `AddItemViewController` that the `ChecklistsViewController` is now its delegate. The proper place to do that when you're using Storyboards is in the `prepareForSegue` method.

» Add this method to ChecklistsViewController.m:

ChecklistsViewController.m

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"AddItem"]) {
    UINavigationController *navigationController = segue.↵
        destinationViewController;
    AddItemViewController *controller = (AddItemViewController *)↵
        navigationController.topViewController;
    controller.delegate = self;
  }
}
```

The `prepareForSegue` method is invoked by UIKit when a segue from one screen to another is about to be performed. It allows us to give data to the new view controller before it will be displayed. Usually you'll do that by setting its properties.

The new view controller can be found in `segue.destinationViewController`. In our app, that is not `AddItemViewController` but the navigation controller that embeds it. To get the `AddItemViewController` object, we can look at the navigation controller's `topViewController` property. This property refers to the screen that is currently active inside the navigation controller.

Once we have a reference to the `AddItemViewController` object, we set its `delegate` property to `self` and the connection is complete. `ChecklistsViewController` is now the delegate of `AddItemViewController`. It took some work, but we're all set now.

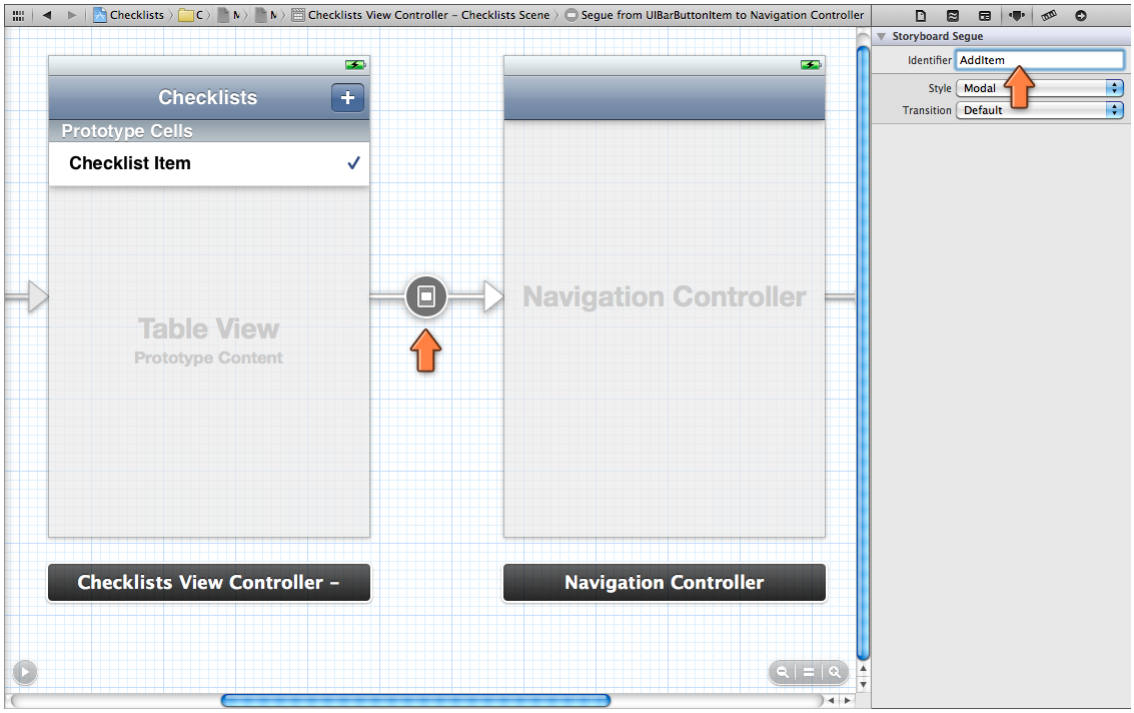
You may wonder why we did this:

```
  if ([segue.identifier isEqualToString:@"AddItem"]) {
    . . .
  }
```

Because there may be more than one segue per view controller, it's a good idea to give each one a unique identifier and to check for that identifier first to make sure you're handling the correct segue.

» Open the Storyboard editor and select the segue between the Checklists View Controller and the Navigation Controller on its right. In the Attributes Inspector, type "AddItem" into the Identifier field:

Naming the segue between the Checklists scene and the navigation controller



» Run the app to see if it works.

Pressing the + button will perform the segue to the Add Item screen with the Checklists screen set as its delegate. When you press Cancel or Done, the `AddItemViewController` sends a message to its delegate (i.e. `ChecklistsViewController`). Currently the delegate simply closes the Add Item screen, but we'll soon make it do more.

Equal or not equal

You may be wondering why we did not use the == operator to check if the segue identifier was equal to the text “AddItem”, in the following manner:

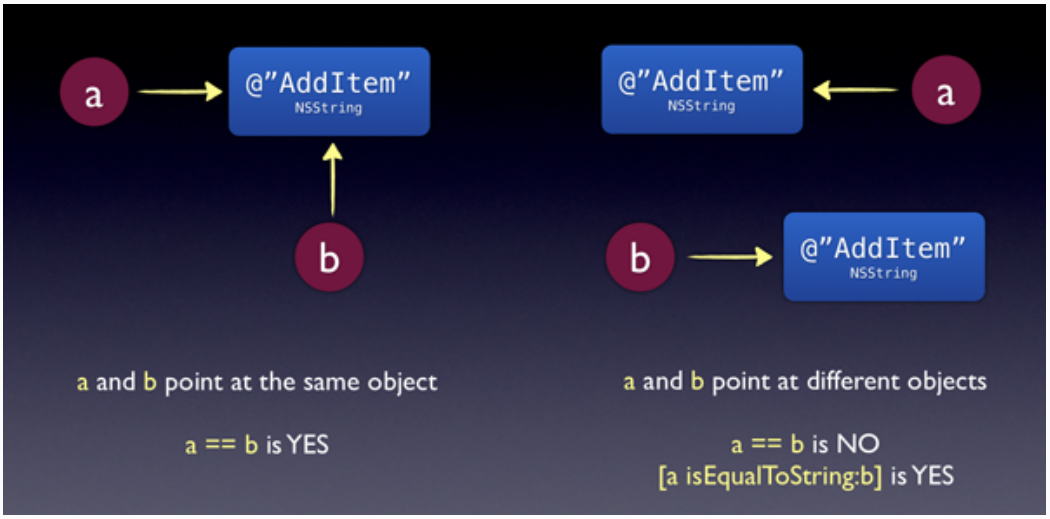
```
if (segue.identifier == @"AddItem") {
  . . .
}
```

This gets into the topic of *object identity*, or what does it mean for two objects to be equal.

If you use ==, you’re checking whether two variables refer to the exact same object. That would be wrong in this case, as the literal text `@"AddItem"` and the value from `segue.identifier` are most likely two separate string objects.

However, both of these string objects may have the same value — they both may contain the text `@"AddItem"`. To compare the values of these objects but not the objects themselves, you use the `isEqual` method, or more specifically for strings, `isEqualToString`.

The difference between using == and isEqualToString to compare two strings



Imagine two people who are both called Joe. They’re different people who just happen to have the same name. If we’d compare them with `joe1 == joe2` then the result would be `NO`, as they’re not the same person. But `[joe1.name isEqualToString:joe2.name]` would be `YES`.

On the other hand, if I’m telling you a story about Joe and this story seems awfully familiar to you, then maybe we happen to know this same Joe. In that case, `joe1 == joe2` would be `YES`.

We can now add the new `ChecklistItem` to our data model and table view. Finally!

» Change the implementation of the `didFinishAddingItem` delegate method to the following:

ChecklistsViewController.m

```
- (void)addItemViewController:(AddItemViewController *)controller ↵
    didFinishAddingItem:(ChecklistItem *)item
{
  int newRowIndex = [items count];
  [items addObject:item];

  NSIndexPath *indexPath = [NSIndexPath indexPathForRow:newRowIndex inSection↵
      :0];
  NSArray *indexPaths = [NSArray arrayWithObject:indexPath];
  [self.tableView insertRowsAtIndexPaths:indexPaths withRowAnimation:↵
      UITableViewRowAnimationAutomatic];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

This is largely the same as what we did in `addItem` before. In fact, I simply copied the contents of the `addItem` action and pasted them into our delegate method. The only difference is that we no longer create the `ChecklistItem` object ourselves; this now happens in the `AddItemViewController`. We merely have to insert this new object into our `items` array. As before, we tell the table view we have a new row for it and then close the Add Items screen.

» Remove the `addItem` action from the ChecklistsViewController.h and .m files as we no longer need this method.

Just to make sure, go into the Storyboard editor and make sure the + button is no longer connected to the `addItem` action. Bad things happen if buttons are connected to methods that no longer exist…

» Run the app and you should be able to add your own items to the list!

You can find the project files for the app up to this point under “04 - Add Item Screen” in the tutorial’s Source Code folder.

Delegates seem like a lot of work!

Why put in all this effort with delegates when you simply could have given `AddItemViewController` a property that points to `ChecklistsViewController`? That would work something like this:

```
// in the .h file
@interface AddItemViewController

@property (nonatomic, weak) ChecklistsViewController *checklistsViewController;

@end

// in the .m file
#import "ChecklistsViewController.h"

@implementation AddItemViewController

@synthesize checklistsViewController;

- (IBAction)done
{
  // directly call a method from ChecklistsViewController
  [checklistsViewController addItemWithText:self.textField.text];
}

@end
```

It's true that this saves some typing, but it also shackles these two objects together. As a general design principle, if screen A launches screen B then you don't want screen B to know too much about the screen that invoked it (A).

When you give `AddItemViewController` a direct pointer to `ChecklistsViewController`, then you can never open the Add Item screen from somewhere else in the app. We won't actually do this in the Checklists app, but it's not uncommon for one screen to be accessible from multiple places. Using a delegate helps to abstract the dependency from screen B on screen A.

In addition, the delegate protocol defines a contract between screen B and any screens that wish to use it. Screen A doesn't have to expose any methods that it might prefer to keep hidden from other objects. By making it a delegate of screen B, screen A doesn't have to put any methods in its public `@interface` beyond those from the delegate protocol.

Anytime you want one part of your app to notify another part about something, usually in order to update the screen, you want to use delegates. It's the iOS way.

Editing existing checklist items

Adding new items to the list is a great step forward for our app, but there are usually three things an app needs to do with data: 1) adding new items (which we've tackled), 2) deleting items (we allow that with swipe-to-delete), and 3) editing existing items (uhh…). The later is useful for when you want to rename an item from your list. We all make typos.

We could make a completely new Edit Item screen but it would work mostly the same as the Add Item screen. The only difference is that it doesn't start out empty but with an existing to-do item.

So instead let's re-use the Add Item screen and make it capable of editing an existing `ChecklistItem` object. When the user presses Done we will update the text in that object and tell the delegate about it so that it can update the label of the corresponding table view cell.

Exercise: Which changes would we need to make to the Add Item screen to enable it to edit existing items? ■

Answer:

1. The screen must be renamed to Edit Item
2. We must be able to give it an existing `ChecklistItem` object
3. We have to place the item's text into the text field
4. When the user presses Done, we should not add a new item object, but update the existing one

There is a bit of a user interface problem, though… How will we actually open the Edit Item screen? In many apps that is done by tapping on the item's row but in our app that already toggles the checkmark. To solve this problem, we'll have to revise the UI of our app a little.

When a row is given two functions, the standard approach is to use a *detail disclosure button* for the secondary task:

The detail disclosure button

Tapping the row itself will still perform the row's main function, in our case toggling the checkmark. But we'll make it so that tapping the disclosure button will open the Edit Item screen.

(An alternative approach is taken by Apple's Reminders app. There the checkmark is on the left and tapping only this left-most section of the row will toggle the checkmark. Tapping anywhere else in the row will bring up the Edit screen for that item. There are also apps that can toggle the whole screen into "Edit mode" and then let you change the text of an item inline. Which solution you choose depends on what works best for your data.)

» Go to the table view cell in the Storyboard editor and in the Attributes Inspector set its Accessory to Detail Disclosure.

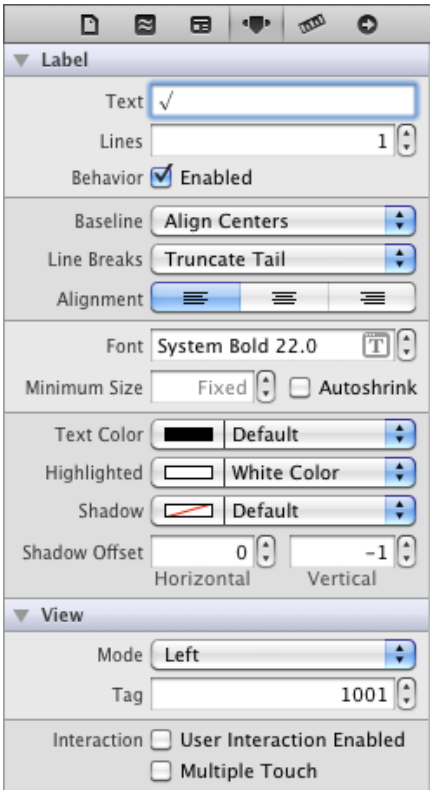
Instead of the checkmark you'll now see a blue chevron (>) button on the right of the cell. This means we'll have to place the checkmark somewhere else.

» Add a new label to the cell to the left of the text label. Give it the following attributes:

- Text: √ (you can type this with Alt/Option+V)
- Font: System Bold, size 22
- Autoshrink: No
- Highlighted color: White
- Tag: 1001

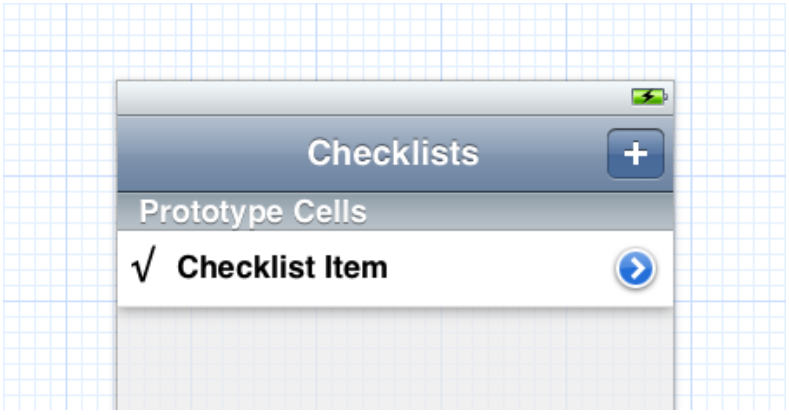
We've give this new label its own tag, so we can easily find it later.

Attributes for the checkmark label



The design of the prototype cell now looks like this:

The new design of the prototype cell



» In ChecklistsViewController.m, change configureCheckmarkForCell to:

ChecklistsViewController.m

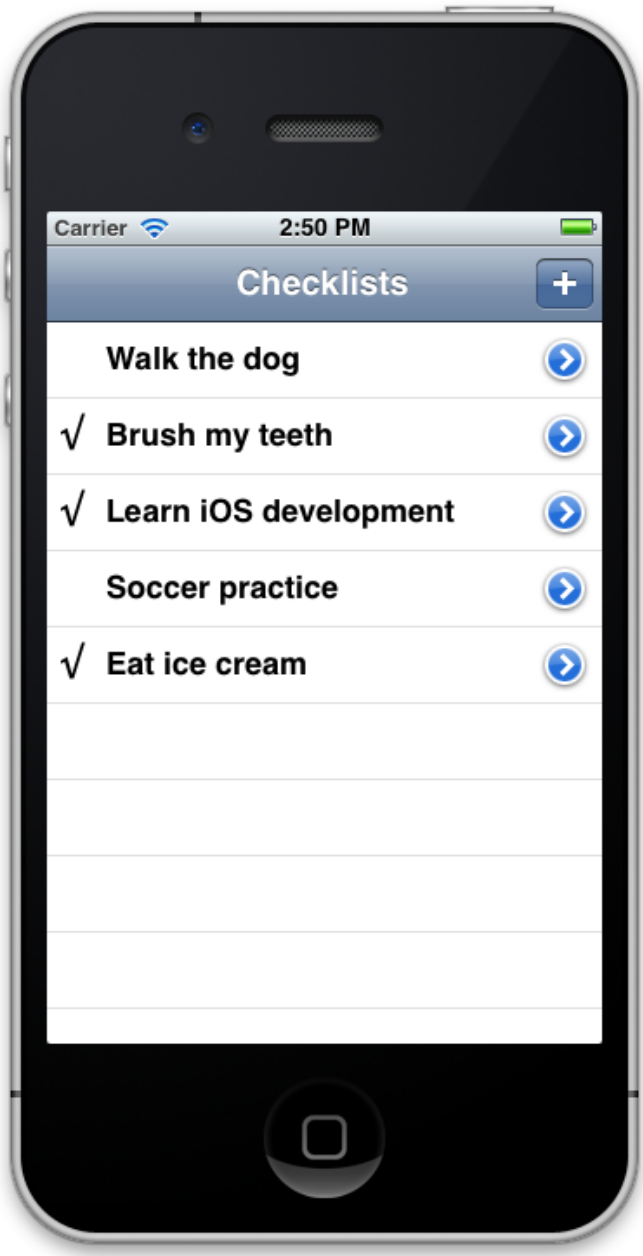
```
- (void)configureCheckmarkForCell:(UITableViewCell *)cell withChecklistItem:(↵
    ChecklistItem *)item
{
  UILabel *label = (UILabel *)[cell viewWithTag:1001];
```

```
  if (item.checked) {
    label.text = @"√";
  } else {
    label.text = @"";
  }
}
```

Instead of changing the cell's `accessoryType` property, this now changes the text in the new label.

» Run the app and you'll see that the checkmark has moved to the left. There is now a blue detail disclosure button on the right. Tapping the row still toggles the checkmark, but tapping the blue button doesn't.

The checkmarks are now on the other side of the cell

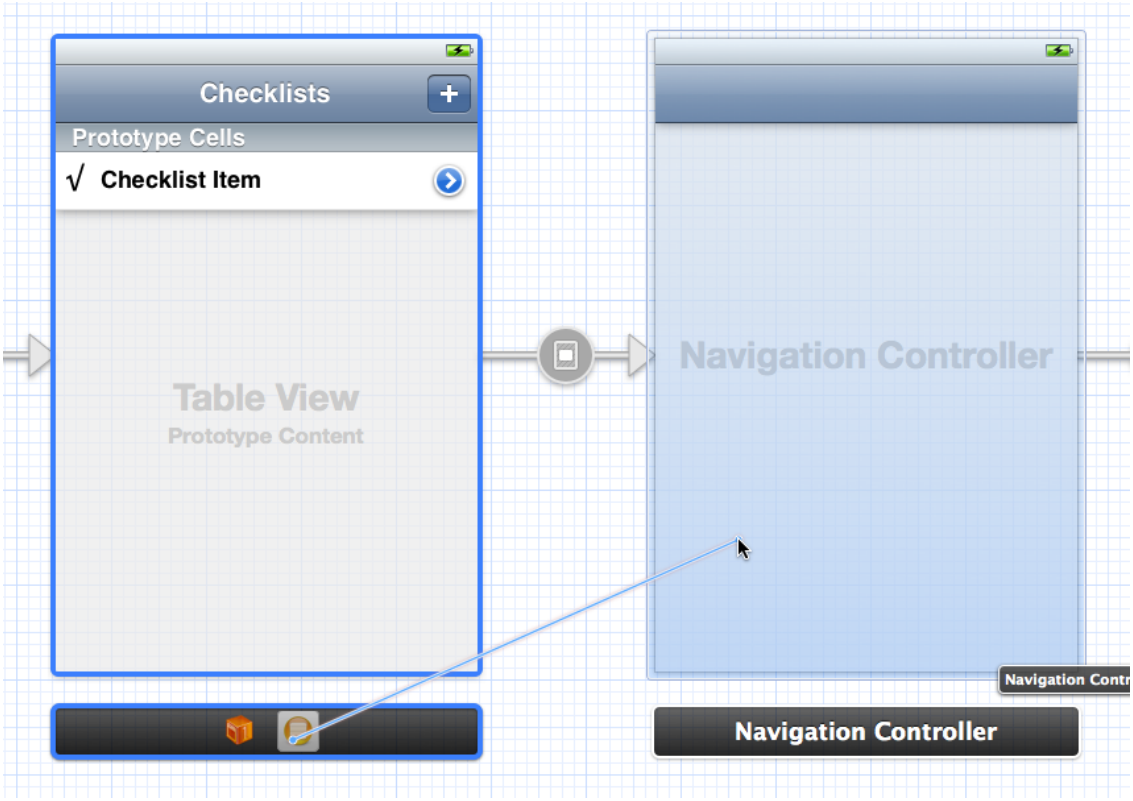


Next we're going to make the detail disclosure button open the Add/Edit Item screen. Unfortunately, the Storyboard editor for iOS 5 doesn't make it possible to create a segue for a disclosure button. If we Ctrl-drag from where the disclosure button is to the Add Item View Controller it attaches a segue to the row itself, not to the disclosure button.

However, we can create a segue that is attached to the table view controller as a whole and then trigger that manually.

» Ctrl-drag from the Checklists view controller icon in the dock area to the Navigation Controller next door and make a Modal segue:

Adding the segue to the view controller



There are now two segues going from the Checklists screen to the navigation controller. For us to make a distinction between the two, they must have unique identifiers.

» Give this new segue the identifier “EditItem” (in the Attributes Inspector).

Note that this new segue isn’t attached to any control. There is nothing on the screen that you can tap or otherwise interact with in order to trigger this segue. We have to perform it programmatically.

» Add the following method to ChecklistsViewController.m. Place it with the other table view delegate methods:

ChecklistsViewController.m

```
- (void)tableView:(UITableView *)tableView ↵
    accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
{
  [self performSegueWithIdentifier:@"EditItem" sender:nil];
}
```

The `accessoryButtonTappedForRowWithIndexPath` method comes from the table view delegate protocol and the name is obvious enough to guess what it does. Inside this method we simply tell the view controller to perform the segue named "EditItem".

If you run the app now, tapping the blue chevron button will open the Add/Edit Item screen. The Cancel and Done buttons won't work because we haven't set the delegate yet. We only set the delegate in `prepareForSegue` for when you tap the + button and perform the "AddItem" segue, not the "EditItem" segue. We'll change that in a minute.

> Note: As of Xcode 4.5 and iOS 6 you can now make segues directly from accessory buttons. When you Ctrl-drag from the table view cell, Xcode asks you whether you want to put the segue on the whole cell or just the accessory button. Because this new feature is not compatible with iOS 5, this tutorial will continue to use the workaround.

Before we fix the delegate business, we shall make the Add/Edit Item screen capable of editing existing `ChecklistItem` objects.

» Add a new property for a `ChecklistItem` object below the other properties in AddItemViewController.h:

AddItemViewController.h

```
@property (nonatomic, strong) ChecklistItem *itemToEdit;
```

This property contains the `ChecklistItem` object that we will be editing. If we're adding a new to-do item, the `itemToEdit` property will be `nil`. That is how the view controller will make the distinction between adding and editing.

» Properties need to be synthesized, so add the following to AddItemViewController.m:

AddItemViewController.m

```
@synthesize itemToEdit;
```

» Change the `viewDidLoad` method to the following:

AddItemViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];
```

```
  if (self.itemToEdit != nil) {
    self.title = @"Edit Item";
    self.textField.text = self.itemToEdit.text;
  }
}
```

Remember that `viewDidLoad` is called when the view controller is created, but before it is shown on the screen. That gives us time to put the user interface in order. In editing mode, when `self.itemToEdit` is not `nil`, we change the title in the navigation bar to "Edit Item". You do this by changing the `self.title` property. The navigation controller looks for this property and automatically changes the title in the navigation bar. We also set the text in the text field to the value from the item's `text` property.

The `AddItemViewController` is now capable of recognizing when it needs to edit an item. If the `self.itemToEdit` property is given a `ChecklistItem` object, then the screen magically changes into the Edit Item screen.

But where do we fill up that `itemToEdit` property? In `prepareForSegue`, of course! That's the ideal place for putting values into the properties of the new screen before it becomes visible.

» Change `prepareForSegue` in ChecklistsViewController.m to the following:

ChecklistsViewController.m

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"AddItem"]) {
    . . .

  } else if ([segue.identifier isEqualToString:@"EditItem"]) {
    UINavigationController *navigationController = segue.↵
        destinationViewController;
    AddItemViewController *controller = (AddItemViewController *)↵
        navigationController.topViewController;
    controller.delegate = self;
    controller.itemToEdit = sender;
  }
}
```

As before, we get the navigation controller from the Storyboard and its embedded `AddItemViewController` using the `topViewController` property. We set the controller's `delegate` property so we're notified when the user taps Cancel or Done.

This is the interesting new bit:

```
  controller.itemToEdit = sender;
```

The `sender` parameter apparently contains the `ChecklistItem` object to edit, but where does that come from?

» Change `accessoryButtonTappedForRowWithIndexPath` to:

ChecklistsViewController.m

```
- (void)tableView:(UITableView *)tableView ↵
    accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
{
  ChecklistItem *item = [items objectAtIndex:indexPath.row];
  [self performSegueWithIdentifier:@"EditItem" sender:item];
}
```

The accessory-tapped method is given the index-path of the row in question. We use that to look up the corresponding `ChecklistItem` object, which we pass to the segue in the `sender` parameter of `performSegueWithIdentifier`.

Normally `sender` will contain a reference to the control that triggered the segue, such as the bar button item or the table view cell that was tapped, but since we're performing the segue manually we can make `sender` anything we want. A `ChecklistItem` object will do just fine, thank you.

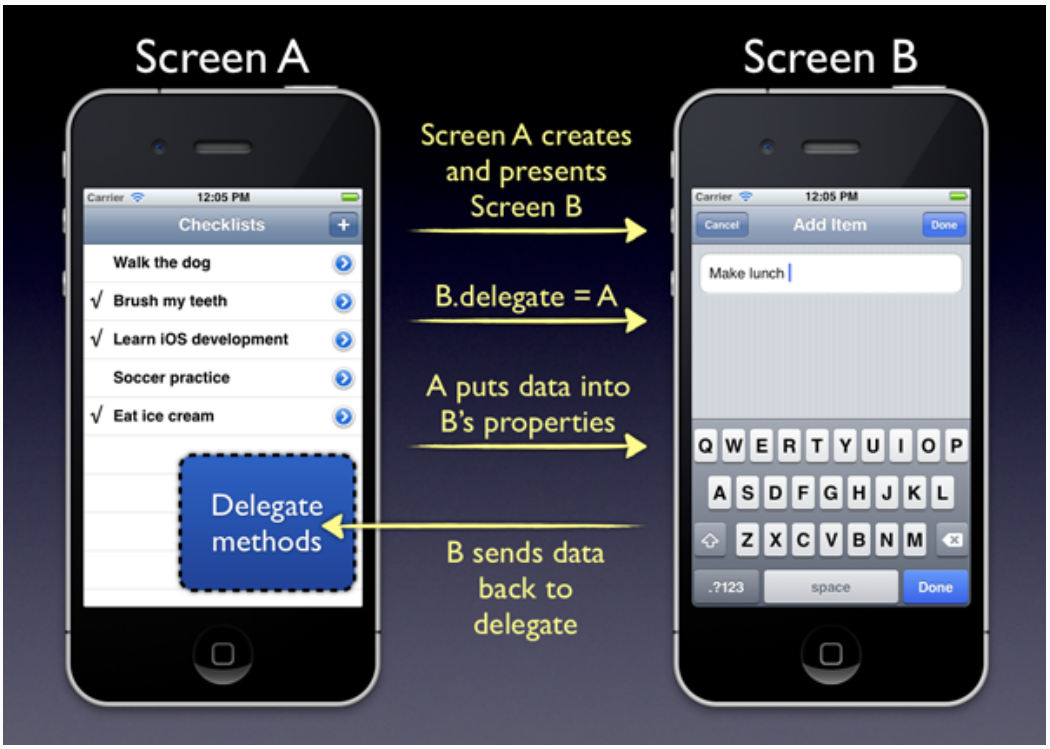
Sending data between the view controllers

We've talked about screen B (the Add/Edit Item screen) passing data back to screen A (the Checklists screen) through the use of delegates. But note that here we're actually passing data from screen A to screen B: the `ChecklistItem` to edit.

This data transfer works two ways: if screen A opens screen B, then A can give B the data it needs. You simply make a property for this data on screen B and then screen A puts something into this property right before it makes screen B visible.

To pass data back from B to A you would use a delegate.

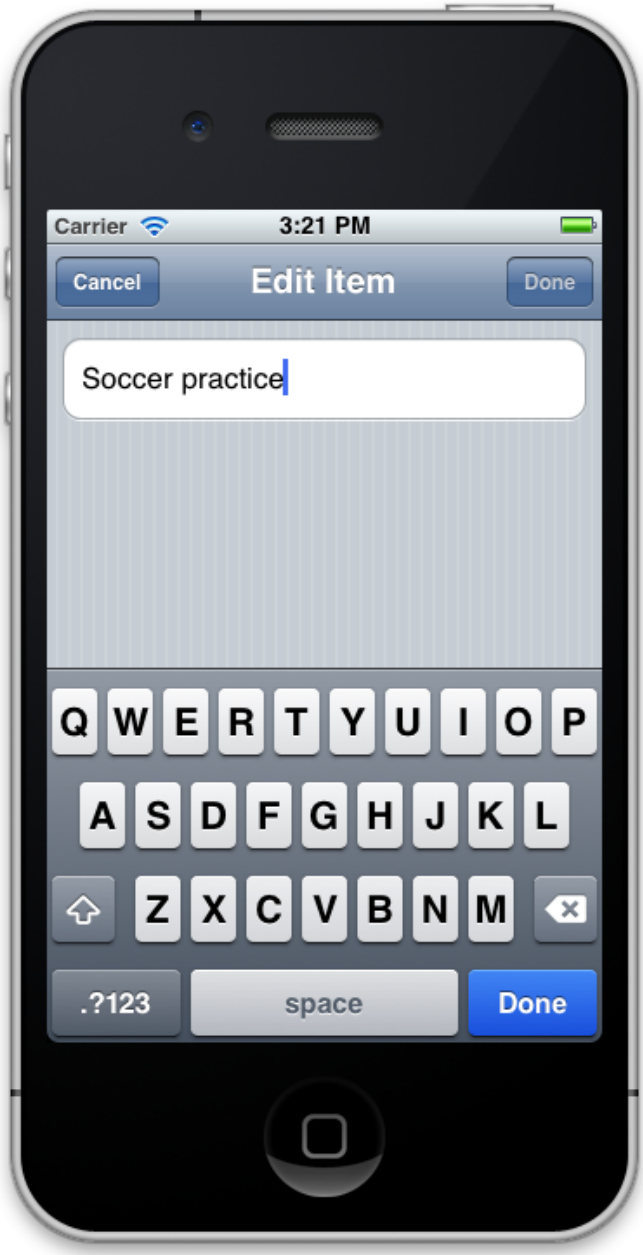
Screen A sends data to B by putting it into B's properties. Screen B sends data back to the delegate.



We're going to do this a few more times in this lesson, just to make sure you don't forget it. :-)

With these steps done, we can now run the app. If you tap the + button, the Add Item screen works as before. However, if you tap the accessory button in an existing row, the screen that opens is now named Edit Item and it already contains the text of that item.

Editing an item



One small problem: the Done button in the navigation bar is initially disabled. This is because we set it disabled in the Storyboard editor.

» Change `viewDidLoad` to fix this:

AddItemViewController.m

```
- (void)viewDidLoad
{
```

```
  [super viewDidLoad];

  if (self.itemToEdit != nil) {
    self.title = @"Edit Item";
    self.textField.text = self.itemToEdit.text;
    self.doneBarButton.enabled = YES;
  }
}
```

We can simply always enable the Done button because when we're editing an existing item we are guaranteed to pass in a text that is not empty.

Our problems don't end here, though. Run the app, tap a row to edit it, and press Done. Instead of changing the text on the existing item, a new to-do item with the new text is now added to the list. We didn't write the code yet to update our data model, so the delegate always thinks it needs to add a new row.

To solve this we will add a new method to our delegate protocol.

» Add the following line to the `@protocol` section in AddItemViewController.h:

AddItemViewController.h

```
- (void)addItemViewController:(AddItemViewController *)controller ↵
    didFinishEditingItem:(ChecklistItem *)item;
```

The full protocol now looks like this:

AddItemViewController.h

```
@protocol AddItemViewControllerDelegate <NSObject>
- (void)addItemViewControllerDidCancel:(AddItemViewController *)controller;
- (void)addItemViewController:(AddItemViewController *)controller ↵
    didFinishAddingItem:(ChecklistItem *)item;
- (void)addItemViewController:(AddItemViewController *)controller ↵
    didFinishEditingItem:(ChecklistItem *)item;
@end
```

We have a method that is invoked when the user presses Cancel and two methods for when the user presses Done. If we're adding a new item, `didFinishAddingItem` is called, but if we were editing an existing item, `didFinishEditingItem` is called. By using different methods the delegate (our view controller) can make a distinction between those two situations.

» In AddItemViewController.m, change the `done` method to:

AddItemViewController.m

```
- (IBAction)done
{
  if (self.itemToEdit == nil) {
    ChecklistItem *item = [[ChecklistItem alloc] init];
    item.text = self.textField.text;
    item.checked = NO;

    [self.delegate addItemViewController:self didFinishAddingItem:item];
  } else {
    self.itemToEdit.text = self.textField.text;
    [self.delegate addItemViewController:self didFinishEditingItem:self.↵
        itemToEdit];
  }
}
```

First we check whether the `itemToEdit` property contains an object. If not, then we're adding a new item and we do the stuff we did before. The new part is this:

```
self.itemToEdit.text = self.textField.text;
[self.delegate addItemViewController:self didFinishEditingItem:self.itemToEdit↵
    ];
```

Nothing too spectacular here. We put the text from the text field into the `ChecklistItem` object from `itemToEdit` and then call the new delegate method.

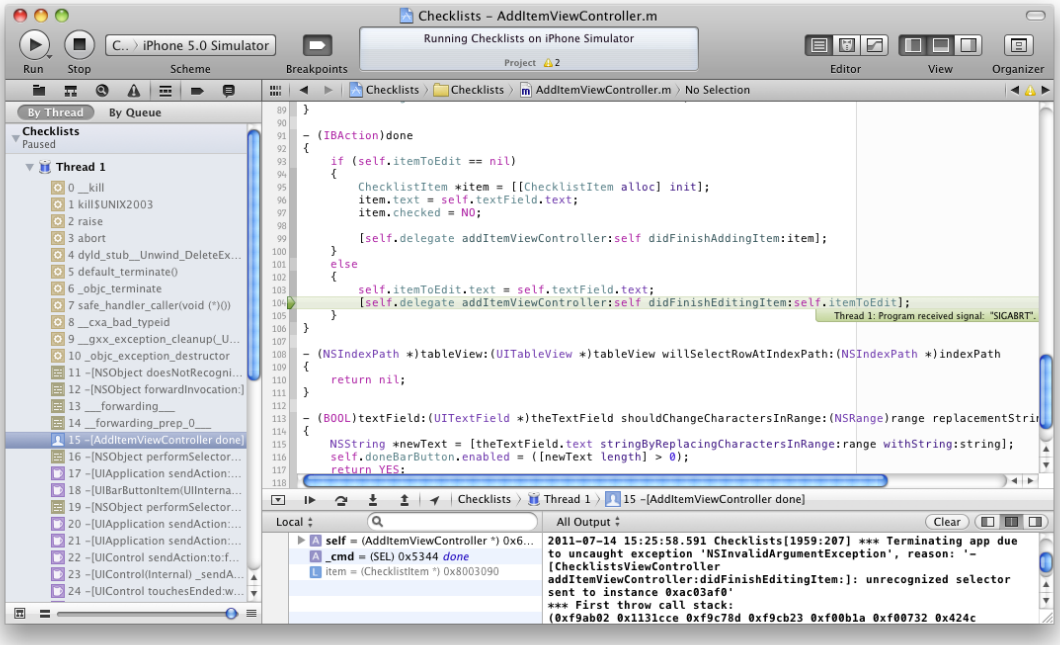
(Note that we pass along the object from `itemToEdit` with the delegate method. This isn't strictly necessary as the delegate could simply look at the `itemToEdit` property to find that object. I chose to do it this way to be consistent with `didFinishAddingItem` because that message also sends along a `ChecklistItem` object.)

If you were to run the app now, it will crash. (Go ahead, try it out.) Once you press Done, the following will appear in your Debug Area:

```
*** Terminating app due to uncaught exception 'NSInvalidArgumentException',
reason: '-[ChecklistsViewController addItemViewController:
didFinishEditingItem:]: unrecognized selector sent to instance 0xab04950'
```

The Xcode window has switched to the debugger and points out which line caused the crash:

The app crashes on the code we just added



It's also possible that Xcode points at the main.m source file as the cause for the crash, but that's a little misleading:

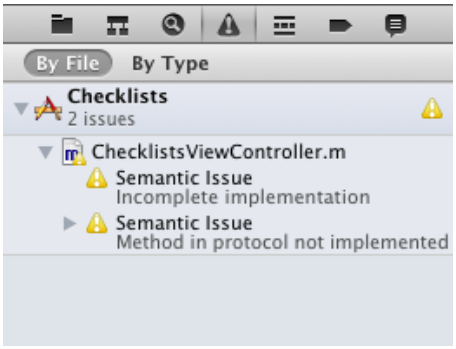
Xcode isn't being very helpful here

```
#import <UIKit/UIKit.h>

#import "ChecklistsAppDelegate.h"

int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil, NSStringFromClass([ChecklistsAppDelegate class]));
    }
}
```

Thread 1: signal SIGABRT

More about debugging later, but I wanted to show you this error as it is very common to get this when you're using delegates. Xcode actually warned us about this problem in the Issue Navigator:

Xcode warns about incomplete implementation



Xcode complains that `ChecklistsViewController` has an "incomplete implementation" because a method in a protocol wasn't implemented. That is also the reason for the crash: unrecognized selector sent.

A *selector* is term Objective-C uses for the name of a method, so this warning means we tried to call a method named `addItemViewController:didFinishEditingItem:` that doesn't actually exist anywhere.

That is not so strange because we only added this method to our delegate protocol but did not actually tell the view controller, the actual delegate, what to do with it.

» Add the following to ChecklistsViewController.m and the crash will be history:

ChecklistsViewController.m

```
- (void)addItemViewController:(AddItemViewController *)controller ↵
    didFinishEditingItem:(ChecklistItem *)item
{
  int index = [items indexOfObject:item];
  NSIndexPath *indexPath = [NSIndexPath indexPathForRow:index inSection:0];
  UITableViewCell *cell = [self.tableView cellForRowAtIndexPath:indexPath];
  [self configureTextForCell:cell withChecklistItem:item];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

The `AddItemViewController` already updated the `ChecklistItem` object for us with the new text but we still need to update the table view. We look for the corresponding cell in the table and tell it to update its label using the `configureTextForCell` method we wrote earlier.

» Run the app again and verify that it now works.

Refactoring the code

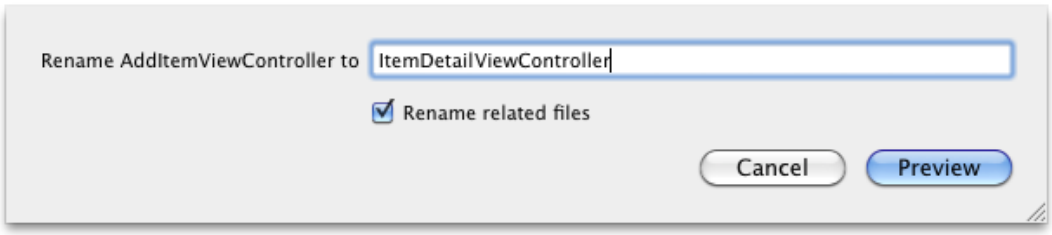
So now we have an app that shows a list of items. You can toggle the checkmark on and off by tapping the rows. You can add new items and edit existing items using the Add/Edit Item screen and you can delete items by swiping the rows. Pretty sweet.

Given the recent changes, I don't think the name `AddItemViewController` is appropriate anymore as this screen is now used to both add items and edit items. I propose we rename it to `ItemDetailViewController`.

» Go to AddItemViewController.h and click in the `@interface` line so that the blinking cursor is on the word `AddItemViewController`. From the Xcode menubar at the top of the screen choose Edit → Refactor → Rename…

Xcode will now ask you for the new name:

Changing the name of AddItemViewController



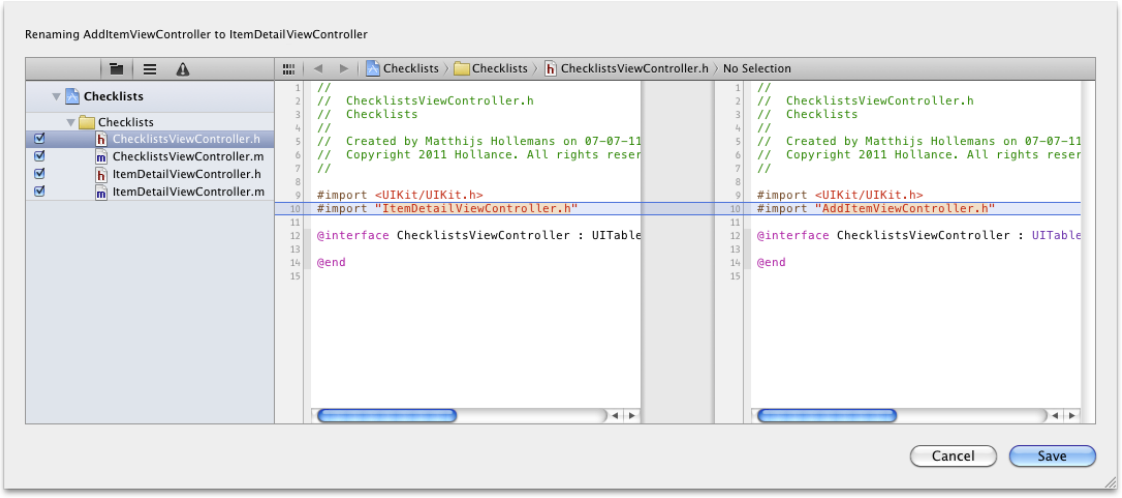
» Type "ItemDetailViewController" for the new name and check "Rename related files".

Tip: If Xcode gives the error message "Wait for indexing and try again", then stop the running app first.

» Press Preview. Xcode opens a screen that shows the files that will change. Click on a filename to see the changes that will be made inside of that file.

You'll see that Xcode will simply rename everything from `AddItemViewController` to `ItemDetailViewController`. It's always smart to check what Xcode is going to do.

Xcode gives a preview of which files it will change



» Click Save to let Xcode do its thing.

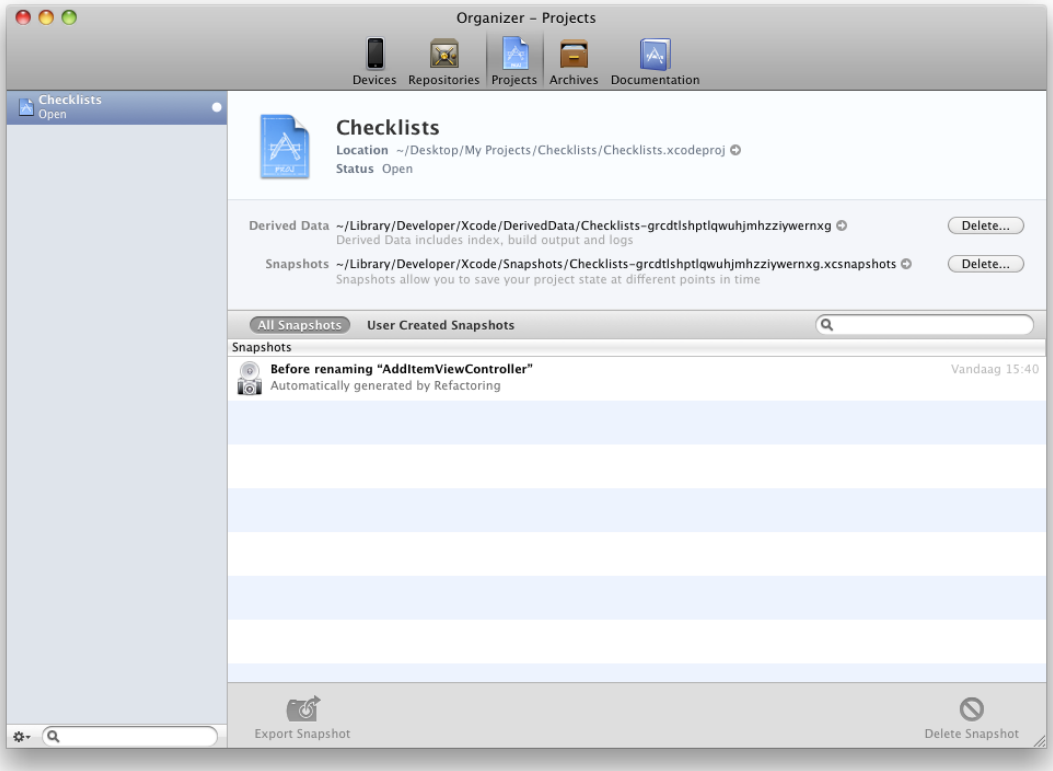
Xcode will now ask you if you want to enable automatic snapshots. A snapshot is a copy of your entire project for safekeeping. It is probably a good idea to enable this. If something goes wrong, you can always go back to an earlier snapshot.

Enable automatic snapshots for simple backups of your project

» Click Enable and wait a few seconds for Xcode to complete the operation.

In case you're curious (or something went wrong!) you can find the snapshots in the Organizer window, under the Projects tab:

The Organizer window lists the snapshots for your project



You can also use the File → Restore Snapshot… option. If at some point you wish to make a manual snapshot, use File → Create Snapshot.

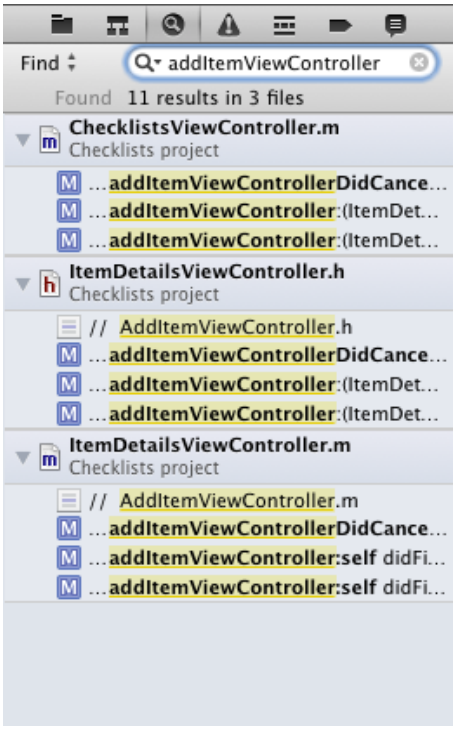
We're not done yet with our refactorings.

» Repeat this process to rename the `AddItemViewControllerDelegate` protocol to `ItemDetailViewControllerDelegate`. You better get used to those long names!

Unfortunately, Xcode isn't able to automatically rename the methods from our delegate protocol for us, so we'll have to do this manually.

» Switch to the Search Navigator and type: "addItemViewController" (without the quotes). This searches through our entire project for that text.

Using the Search navigator to find the methods to change



» You will have to change:

- `addItemViewControllerDidCancel:` into `itemDetailViewControllerDidCancel:`
- `addItemViewController:` into `itemDetailViewController:`

After these changes the `@protocol` in ItemDetailViewController.h now has these methods:

ItemDetailViewController.h

```
- (void)itemDetailViewControllerDidCancel:(ItemDetailViewController *)↵
    controller;
- (void)itemDetailViewController:(ItemDetailViewController *)controller ↵
    didFinishAddingItem:(ChecklistItem *)item;
- (void)itemDetailViewController:(ItemDetailViewController *)controller ↵
    didFinishEditingItem:(ChecklistItem *)item;
```

» You'll also need to change these method names in ItemDetailViewController.m and ChecklistsViewController.m.

I always repeat the search afterwards to make sure I didn't skip anything by accident.

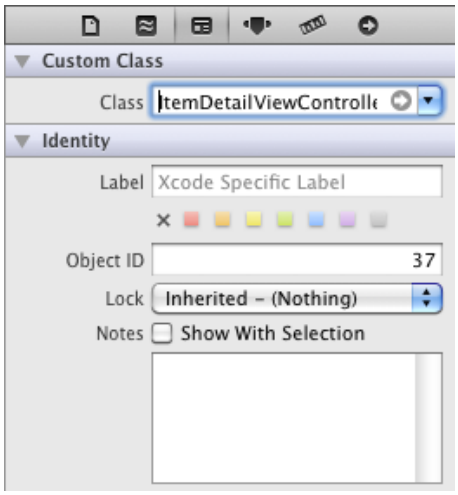
There is one more thing we need to verify. Depending on your version of Xcode, the Refactor tool may not be smart enough to also change the view controller inside the Sto-

ryboard. If that happens, the Storyboard still thinks our view controller is named `AddItemViewController` and the app will crash with the following message:

```
Checklists[2248:207] Unknown class AddItemViewController in Interface Builder ↵
    file.
```

» Go to the Storyboard Editor and select the Add Item View Controller. In the Identity Inspector, change Class to "ItemDetailViewController":

Don't forget to change the Storyboard as well



Because we made quite a few changes all over the place, I always like to do a clean build just to make sure Xcode picks up all these changes and that there are no more warnings or compiler errors. You don't have to be paranoid about this, but it's good practice to do a clean build once in a while.

» From Xcode's menubar choose Product → Clean. When the clean is done choose Product → Build (or simply press the Run button). If there are no issues, then run the app again and test the various features just to make sure everything still works!

Tip: If renaming gives you problems, then double-check your spelling. Objective-C is case-sensitive, so it considers "itemDetailViewController" and "ItemDetailViewController" to be two completely different words.

You can find the project files for the app up to this point under "05 - Edit Items" in the tutorial's Source Code folder.

Iterative development

If you think this approach to development is a little messy, then you're absolutely right. We started out with one design but as we were developing it we found out that things didn't work out so well in practice and that we had to refactor our approach a few times to find a way that works. Well, this is how software development goes in practice.

You build a small part of your app and everything looks and works fine. Then you add the next small part on top of that and suddenly everything breaks down. The proper thing to do is to go back and restructure your entire approach so that everything is hunkey-dorey again… Until the next change you need to make.

Software development is a constant process of refinement. In these tutorials I didn't want to just give you a perfect piece of code and explain to you how each part works. That's not how software development happens in the real world. So instead, we're working our way from zero to a full app, exactly the way a pro developer would, including the mistakes and dead ends.

Isn't it possible to create a design up-front (sometimes called a "software architecture design") that deals with all of these situations, like a blueprint but for software? I don't believe in such designs. Sure, it's always good to plan ahead. Before writing this chapter, I made a few quick sketches of how I imagined the app would turn out. That was useful to envision the amount of work, but as usual some of my assumptions and guesses turned out to be wrong and the design stopped being useful about halfway in. And this is only a simple app!

That doesn't mean you shouldn't spend any time on planning and design, just not too much. ;-) Simply start somewhere and keep going until you get stuck, then backtrack and improve on your approach. This is called *iterative development* and it's usually faster than meticulous up-front planning and provides better results.

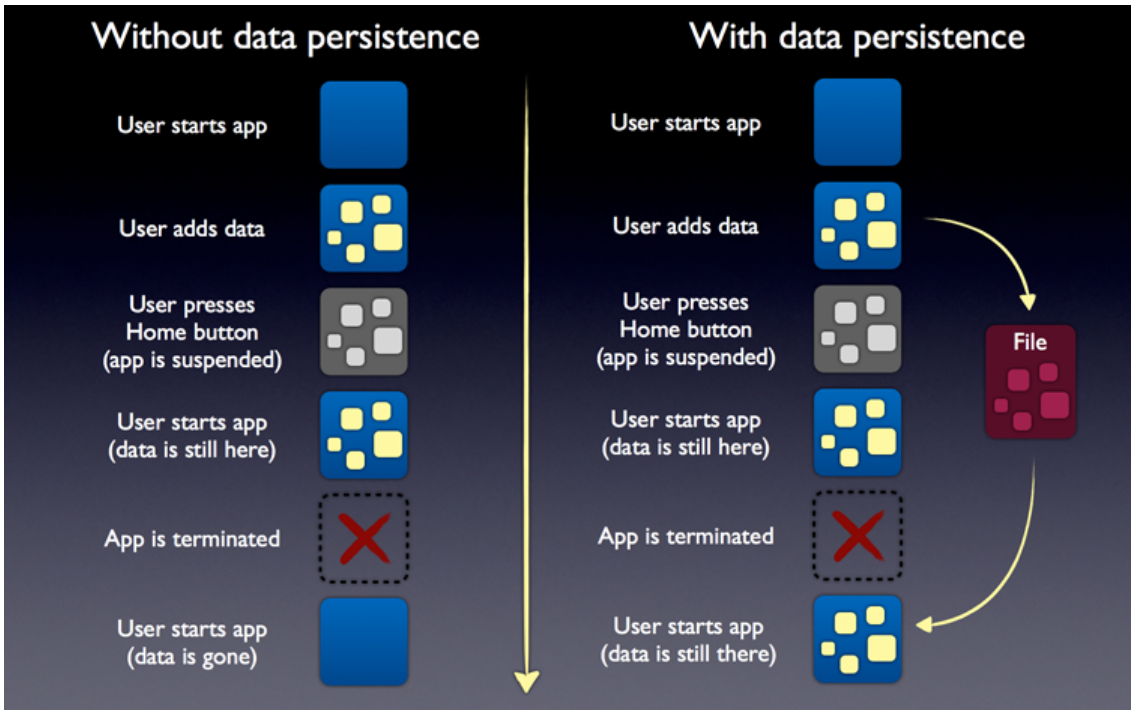
Saving and loading the checklist items

Any new to-do items that you add to the list cease to exist when you terminate the app (by pressing the Stop button in Xcode, for example). You can also delete the five default items that we put in the list but they keep reappearing after a new launch. That's not how a real app should behave, of course.

Thanks to the multitasking features of iOS 4 and up, an app stays in memory when you close it. The app goes into a suspended state where it does absolutely nothing but will still hang on to its data. During normal usage, users will never truly terminate the app, just suspend it. However, the app can still be terminated when the iPhone runs out of available working memory as iOS will terminate any suspended apps in order to free up memory when necessary. And if they really want to, users can kill apps by hand or reset their entire device.

Just keeping the list of items in memory is not good enough because there is no guarantee that the app will remain in memory forever, whether active or suspended. Instead, we will need to *persist* this data in a file on the iPhone's long-term flash storage. This is no different than saving a file from your word processor on your desktop computer except that iPhone apps should take care of this saving automatically. The user shouldn't have to press a Save button just to make sure unsaved data is safely placed in long-term storage.

Apps need to persist data just in case the app is terminated



In this section we will:

- Determine where in the file system we can place the file that will remember the to-do list items.
- Save the to-do items to that file whenever the user changes something: adds a new item, toggles a checkmark, et cetera.
- Load the to-do items from that file when the app starts up again after it was terminated.

Let's get crackin'!

The Documents directory

iOS apps live in a sheltered environment, also known as the *sandbox*. Each app has its own directory for storing files but cannot access the directories or files of any other apps. This is a security measure, designed to prevent malicious software such as viruses from doing any damage. If an app can only change its own files, then it cannot break any other part of the system.

Your apps can store files in the so-called "Documents" directory. You are guaranteed this directory is always available in the app's sandbox. The contents of the Documents directory are backed up when the user syncs their device with iTunes. When you release a new version of your app and users install the update, the contents of the Documents folder are left untouched. So any data the app has saved into this folder stays there even if the app is updated.

Let's look at how this works.

» Add the following methods to `ChecklistsViewController`, above `viewDidLoad`:

ChecklistsViewController.m

```
- (NSString *)documentsDirectory
{
  NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, ↵
      NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths objectAtIndex:0];
  return documentsDirectory;
}

- (NSString *)dataFilePath
{
  return [[self documentsDirectory] stringByAppendingPathComponent:@"Checklists↵
      .plist"];
}
```

The `documentsDirectory` method is something I added for convenience. There is no standard method we can call to get the full path to the Documents folder, so I rolled my own.

The `dataFilePath` method uses `documentsDirectory` to construct the full path to the file that we will use to store the checklist items. This file is named "Checklists.plist" and it lives inside the Documents directory.

We use the `NSString` method `stringByAppendingPathComponent` to build a proper file system path to Checklists.plist. We can call this method because the return value of `[self documentsDirectory]` is also an `NSString`. Recall that `NSString` objects are immutable, so this method will create a new string object with the full path to our file.

You could also have constructed the full path like this:

```
- (NSString *)dataFilePath
{
  return [NSString stringWithFormat:@"%@/Checklists.plist", [self ↵
      documentsDirectory]];
}
```

This adds "Checklist.plist" to the Documents directory path, separated by a forward slash. However, I prefer to use `stringByAppendingPathComponent` since that frees me from worrying about whether to use a forward slash or a backward slash, what to do if there already is a slash in the folder name, and many other tiny concerns. The built-in objects from iOS come with a lot of useful helper methods like these and it's often better to use them instead of trying to do things on your own.

» Add the following two `NSLog()` statements to `viewDidLoad`:

ChecklistsViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  NSLog(@"Documents folder is %@", [self documentsDirectory]);
  NSLog(@"Data file path is %@", [self dataFilePath]);

  . . .
}
```

» Run the app. Xcode's Debug Area will now show you where your app's Documents directory is actually located.

If I run the app from the Simulator, on my system it says:

```
Documents folder is /Users/matthijs/Library/Application Support/
iPhone Simulator/5.0/Applications/
21BAB5FA-EE73-4728-9B1C-2B9829E9E146/Documents

Data file path is /Users/matthijs/Library/Application Support/
iPhone Simulator/5.0/Applications/
21BAB5FA-EE73-4728-9B1C-2B9829E9E146/Documents/Checklists.plist
```

If you run it on your iPhone, the path will look somewhat different. Here's what mine says (this is on an iPod touch):

```
Documents folder is
/var/mobile/Applications/FDD50B54-9383-4DCC-9C19-C3DEBC1A96FE/Documents

Data file path is /var/mobile/Applications/FDD50B54-9383-4DCC-9C19-C3DEBC1A96FE
/Documents/Checklists.plist
```

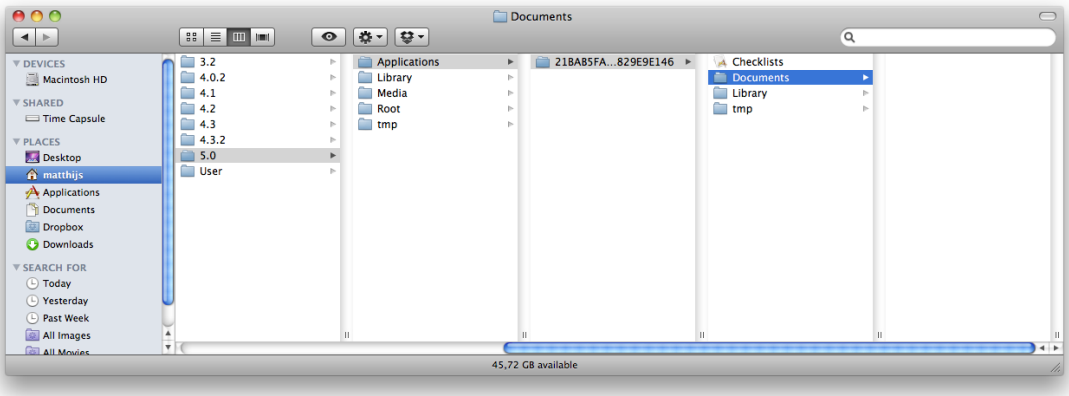
The name of the folder that contains the app is "21BAB5FA-EE73-4728-9B1C-2B9829E9E146" (on the Simulator) and "FDD50B54-9383-4DCC-9C19-C3DEBC1A96FE" (on the device). This is a random ID that Xcode (or iTunes) picks when it installs the app on the Simulator or the device. Anything inside that folder is part of the app's sandbox.

For the rest of this section, run the app on the Simulator instead of a device. That makes it easier to look at the files we'll be writing. Because the Simulator stores the app's files in a regular folder on your Mac, we can easily examine them from Finder.

» Open a new Finder window by clicking on the Desktop and typing Cmd+N. Then press Cmd+Shift+G and paste the full path to the Documents folder in the dialog.

The Finder window will go to that folder. Keep this window open so you can see that the Checklists.plist file is actually being created when we get to that part.

The app's directory structure in the Simulator



Tip: Are you using OS X Lion (10.7) or Mountain Lion (10.8) and you cannot find the Library folder in your home directory? To fix this, open Terminal and type the following command:

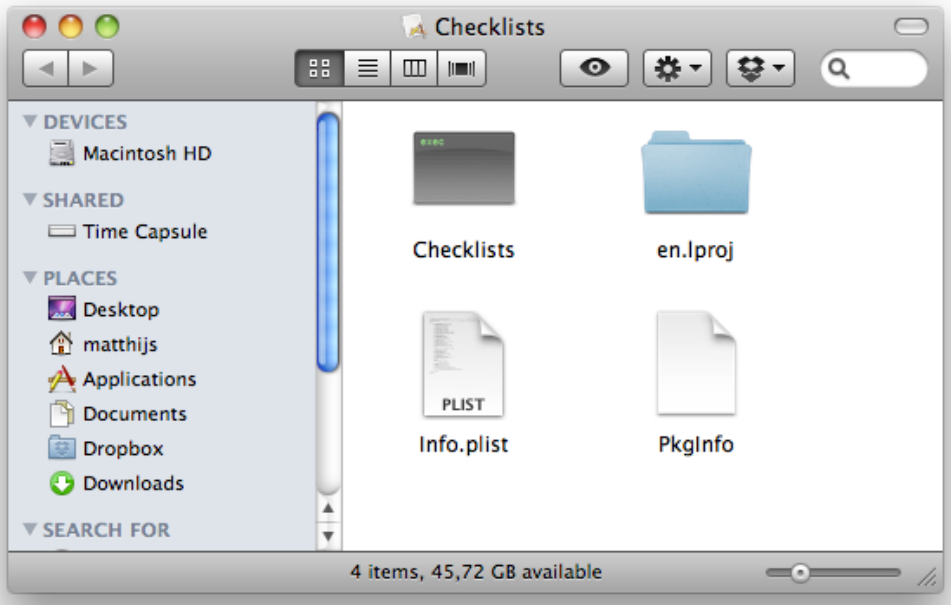
```
chflags nohidden ~/Library
```

You can see several things inside the app's directory:

- Checklists, which is our application bundle.
- The Documents directory where we'll put our data files. Currently the Documents folder is still empty.
- The Library directory has cache files and preferences files. The contents of this directory are managed by the operating system.
- The tmp directory. This is for temporary files. Sometimes apps need to create files for temporary usage. You don't want these to clutter up your Documents folder, so tmp is a good place to put them. The OS will clear out this folder from time to time.

If you're curious about what is exactly in the application bundle, then right click its file-name and choose Show Package Contents. The app bundle is really a folder although Finder pretends it isn't. With Show Package Contents you can see what is inside that folder.

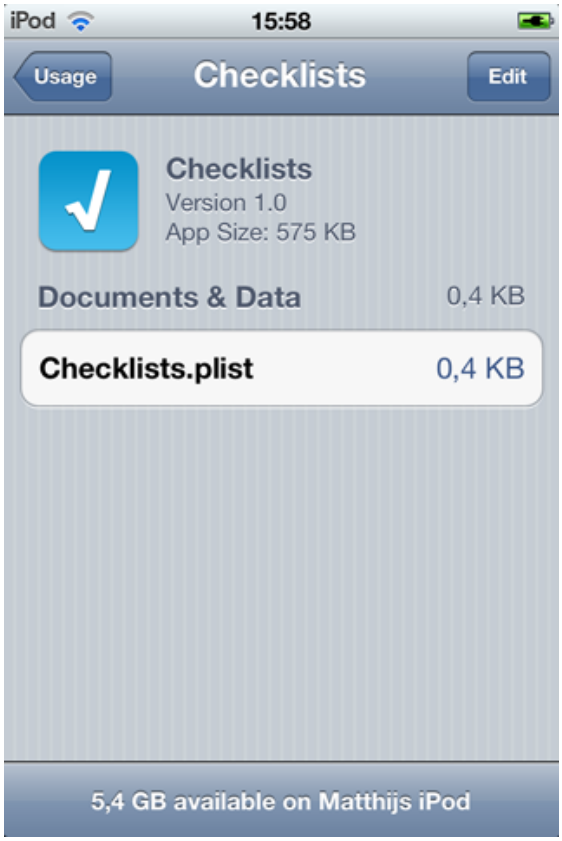
The contents of our application bundle



Later we'll discuss in more detail what is going on here.

As of iOS 5 it is also possible to look inside the Documents directory of apps on your device. On your iPhone or iPod, go to Settings → General → Usage and tap the name of an app. You'll now see the contents of its Documents folder:

Viewing the Documents folder on the device



Saving the checklist items

Back to our app. We are going to write some code that will save the list of checklist items to the Checklists.plist file when the user adds a new item or edits an existing item. Once we are able to save the items we will add the code that is required to load this list.

So what is a .plist file? You've already seen an Info.plist in the Bull's Eye lesson. All apps have one, including our Checklists app (see the Project Navigator for the file named Checklists-Info.plist). Plist stands for Property List and is an XML file format that stores structured data, usually in a form similar to the Info.plist file (a list of settings and their values). Property List files are very common in iOS, are suitable for many types of data storage and best of all, they are simple to use. I like that!

To save our checklist items we'll use the `NSCoder` system, which lets objects store their data in a structured file format. We actually don't really care much about that file format. In this case it happens to be a .plist file but we're not directly going to mess with that file. All we care about is that the data gets stored in some kind of file in our app's Documents folder, but we'll leave the technical details for `NSCoder` to deal with.

We've already used `NSCoder` behind the scenes because that's exactly how nibs and Storyboards work. When you create nib for a view controller or add a view controller to a Storyboard, Xcode uses the `NSCoder` system to write this object to a file (encoding). Then when your application starts up, it uses `NSCoder` again to read the objects from the nib or Storyboard file (decoding). This process of converting objects to files and back again is also known as *serialization*.

I like to think of this whole process as freezing objects. We take a living object and freeze it so that it is now suspended in time. We store that frozen object into a file where it will spend some time in cryostasis. Later we can read that file into memory and defrost the object to bring it back to life again.

» Add the following method to ChecklistsViewController.m, below `dataFilePath`:

ChecklistsViewController.m

```
- (void)saveChecklistItems
{
  NSMutableData *data = [[NSMutableData alloc] init];
  NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc] ↵
      initForWritingWithMutableData:data];
  [archiver encodeObject:items forKey:@"ChecklistItems"];
  [archiver finishEncoding];
  [data writeToFile:[self dataFilePath] atomically:YES];
}
```

This method takes the contents of our `items` array and in two steps converts this to a block of binary data and then writes this data to a file:

1. `NSKeyedArchiver`, which is a form of `NSCoder` that creates plist files, encodes the array and all the `ChecklistItems` in it into some sort of binary data format that can be written to a file.
2. That data is placed in an `NSMutableData` object, which will write itself to the file specified by `dataFilePath`.

It's not really important that you understand how `NSKeyedArchiver` works internally. The format that it stores the data in isn't of great significance to us. All we care about is that it allows us to put our objects into a file and read them back later.

We have to call the `saveChecklistItems` method whenever the list of items is modified, which is in the `ItemDetailViewControllerDelegate` methods.

» Make the following changes in these methods:

ChecklistsViewController.m

```
- (void)itemDetailViewController:(ItemDetailViewController *)controller ↵
    didFinishAddingItem:(ChecklistItem *)item
{
  . . .

  [self saveChecklistItems];

  [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)itemDetailViewController:(ItemDetailViewController *)controller ↵
    didFinishEditingItem:(ChecklistItem *)item
{
  . . .

  [self saveChecklistItems];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

» Let’s not forget the swipe-to-delete function:

ChecklistsViewController.m

```
- (void)tableView:(UITableView *)tableView commitEditingStyle:(↵
    UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)↵
    indexPath
{
  [items removeObjectAtIndex:indexPath.row];

  [self saveChecklistItems];

  NSArray *indexPaths = [NSArray arrayWithObject:indexPath];
  [tableView deleteRowsAtIndexPaths:indexPaths withRowAnimation:↵
      UITableViewRowAnimationAutomatic];
}
```

» And when we turn the checkmark on a row on or off:

ChecklistsViewController.m

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
     *)indexPath
{
  . . .

  [self saveChecklistItems];
```

```
  [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Just calling `NSKeyedArchiver` on our `items` array is not enough. If you were to run the app now and do something that results in a save, such as tapping a row, the app crashes with the following error:

```
*** Terminating app due to uncaught exception 'NSInvalidArgumentException',
reason: '-[ChecklistItem encodeWithCoder:]: unrecognized selector sent to
instance 0x6a26810'
```

The Xcode debugger will point to this line as the culprit:

```
[archiver encodeObject:items forKey:@"ChecklistItems"];
```

You've already seen the "unrecognized selector" error message before. This means we forgot to implement a certain method. In this case, the missing method appears to be `encodeWithCoder` on the `ChecklistItem` object.

We asked `NSKeyedArchiver` to encode our array of items, so it not only has to encode the array itself but also each `ChecklistItem` object inside that array. `NSKeyedArchiver` knows how to encode an `NSMutableArray` object but it doesn't know anything about `ChecklistItem`. So we have to help it out a bit.

» Change the `@interface` line in ChecklistItem.h:

ChecklistItem.h

```
@interface ChecklistItem : NSObject <NSCoding>
```

Recall that < > means that an object conforms to a protocol. In this case we're adding the `NSCoding` protocol to `ChecklistItem`.

» Add the following to ChecklistItem.m, below the `init` method (if your ChecklistItem.m does not have an `init` method, then simply add this below the `@synthesize` line):

ChecklistItem.m

```
- (void)encodeWithCoder:(NSCoder *)aCoder
{
```

```
  [aCoder encodeObject:self.text forKey:@"Text"];
  [aCoder encodeBool:self.checked forKey:@"Checked"];
}
```

This is the missing method from the unrecognized selector error. When `NSKeyedArchiver` tries to encode the `ChecklistItem` object it will send it an `encodeWithCoder` message.

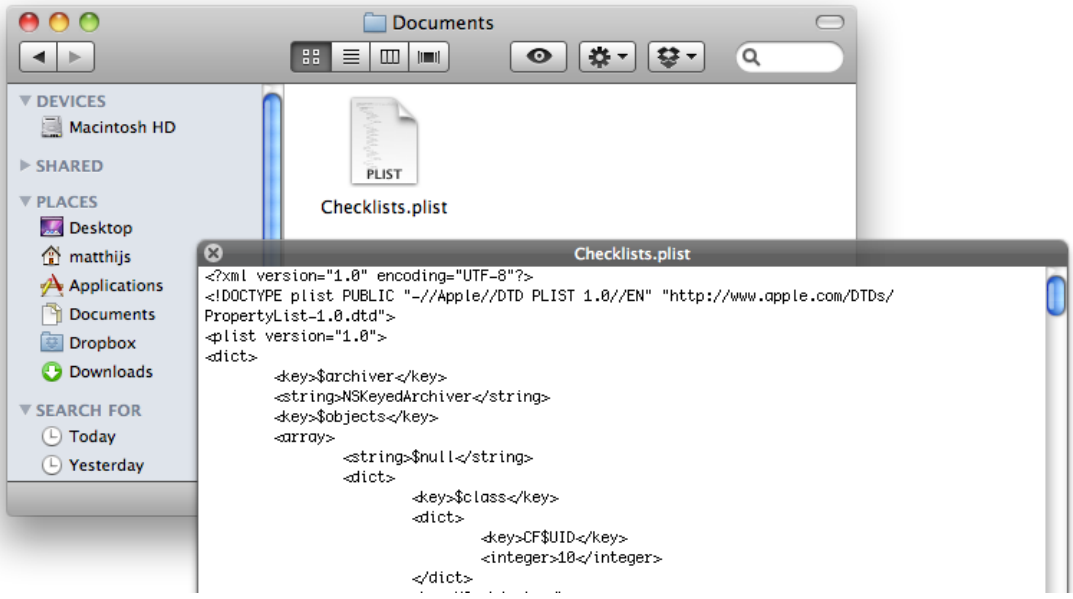
Here we simply say: we have an object named "Text" that contains the value of our property `self.text` and a boolean named "Checked" that contains the value of `self.checked`. Just these two lines are enough to make the coder system work.

» Run the app again and tap a row to toggle a checkmark. The app didn't crash? Good!

Note: At this point Xcode may give a warning that the implementation of `ChecklistItem` is incomplete. That's OK, you will add the missing method shortly.

» Go to the Finder window that has the app's Documents directory open. (If you forgot where that was, it's in the Library folder in your home directory and then Application Support/iPhone Simulator/5.0/Applications/weird number/Documents.)

The Documents directory now contains a Checklists.plist file



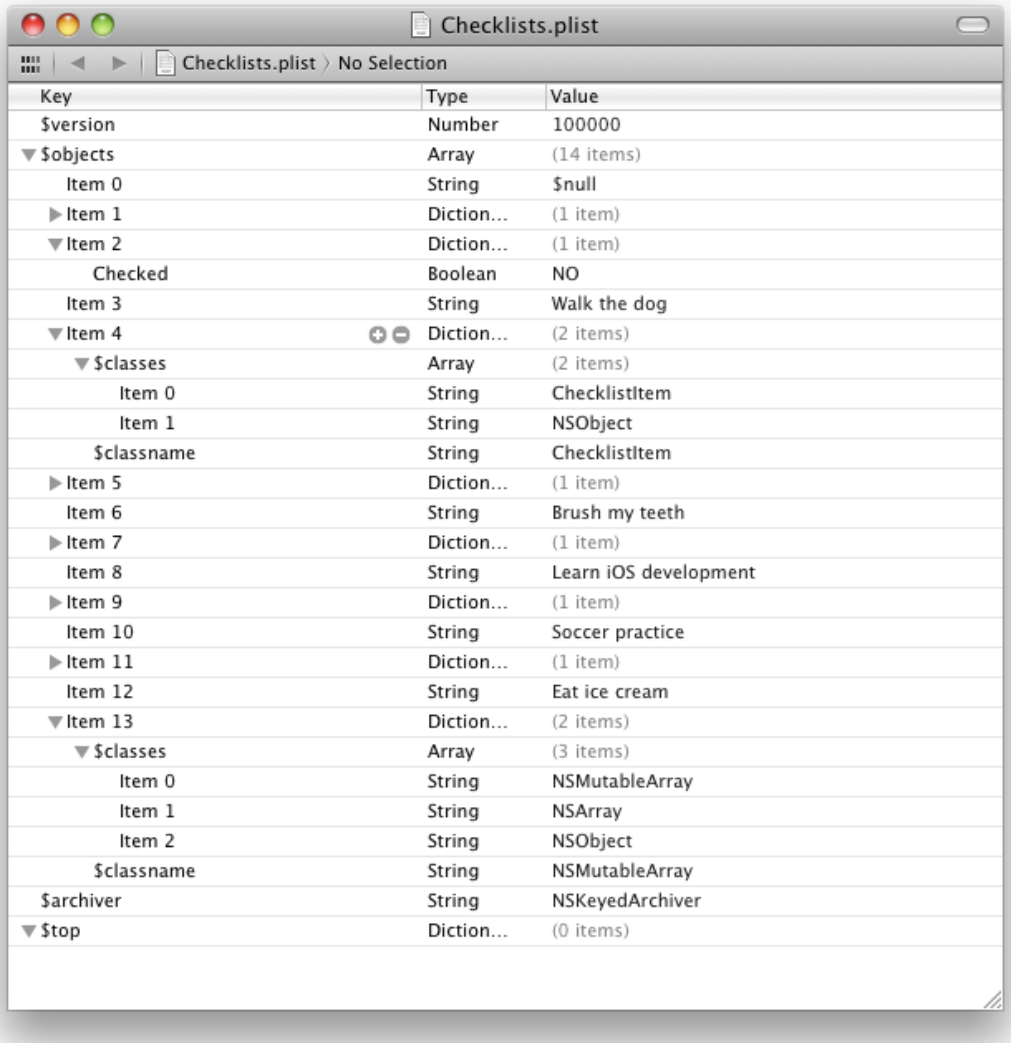
There is now a Checklists.plist file in the Documents folder, which contains the items from our list. You can look inside this file if you want, but the contents won't make much sense to you. Even though it is XML, this file wasn't intended to be read by humans, only by the `NSKeyedArchiver` system.

If you're having trouble viewing the XML it may be because the plist file isn't stored as text but as a binary format. Some text editors support this file format and can read it as if it were text (TextWrangler is a good one and is a free download on the Mac App Store). You can also use Finder's Quick Look feature to view the file. Simply select the file in Finder and press the space bar.

Naturally, you can also open the plist file with Xcode.

» Right-click the Checklists.plist file and choose Open With → Xcode.

Checklist.plist in Xcode



It still won't make much sense but it's fun to look at anyway. If you expand some of the rows you can see that this file was made by `NSKeyedArchiver` and that the names of our

ChecklistItems are also in there. But exactly how everything fits together, I have no idea.

Loading the file

Saving is all well and good but pretty useless by itself so let's implement the loading of the Checklists.plist file. It's very straightforward, we're basically going to do the same thing we just did but in reverse.

You may have noticed Xcode is complaining that ChecklistItem does not implement another method from the NSCoding protocol, initWithCoder. That is the method for unfreezing the objects from the file.

» Add the following directly above the encodeWithCoder method:

ChecklistItem.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super init])) {
    self.text = [aDecoder decodeObjectForKey:@"Text"];
    self.checked = [aDecoder decodeBoolForKey:@"Checked"];
  }
  return self;
}
```

Inside initWithCoder we do the opposite from encodeWithCoder. We take objects from the NSCoder's decoder object and put their values inside our own properties. That's all it takes! What we stored earlier under the "Text" key now goes back into our self.text property. Likewise for the boolean "Checked" value and self.checked.

Loading the Checklists.plist file will be done by an NSKeyedUnarchiver object. That unarchiver does the following behind the scenes to create the ChecklistItem objects:

```
ChecklistItem *item = [[ChecklistItem alloc] initWithCoder:someDecoderObject];
```

When we created the ChecklistItems by hand, we did this:

```
ChecklistItem *item = [[ChecklistItem alloc] init];
```

The difference is that NSKeyedUnarchiver will not use the regular init method but the one named initWithCoder. It is not uncommon for objects to have more than one init

method. They are always named "initSomething", which is a mandatory convention. But which one is used depends on the circumstances.

We use the regular `init` method for creating `ChecklistItem` objects when the user presses the + button and fills in the Add Item screen, and we use `initWithCoder` to restore `ChecklistItems` that were saved to disk.

Init methods

Method names beginning with the word "init" are special in Objective-C. You only use them when you're creating new objects. First you `alloc` the object to reserve a chunk of memory big enough to hold all of the object's data (its instance variables), followed by a call to an `init` method to initialize the object so that it is ready for use.

The implementations of these init methods, whether they're called `init` or `initWithCoder` or something else, always follow the same series of steps. When you write your own init methods, you need to stick to those steps as well.

When we created the ChecklistItem.h and .m source files, Xcode may have already added the following init method for us (depending on your version of Xcode):

```
- (id)init
{
  self = [super init];
  if (self) {
    // Initialization code here.
  }

  return self;
}
```

This is the standard way to write an init method. First you call `[super init]` to initialize this object's superclass and then assign the result to `self`. If you're coming from another programming language and this looks weird to you, well, that's Objective-C for you.

If you haven't done any object-oriented programming at all, then you may not know what a *superclass* is. That's fine, we'll completely ignore this topic until the next tutorial. Just remember that sometimes objects need to send messages to something called `super` and if you forget to do this, bad things are likely to happen.

After assigning the result from `[super init]` to `self`, you have to look at the value of `self` to determine that it is not `nil`:

```
  if (self) {
    // Initialization code here.
  }
```

The statement `if (self)` is shorthand for: `if (self != nil)`.

In the `initWithCoder` method from `ChecklistItem`, I used a common Objective-C idiom:

```
if ((self = [super init])) {
  . . .
}
```

This combines these two lines into one:

```
self = [super init];
if (self != nil) {
  . . .
}
```

If you wanted to be fully explicit, you'd write it as:

```
if ((self = [super init]) != nil) {
  . . .
}
```

This still calls `[super init]`, assigns it to `self` and then checks if `self` is not `nil`, except it does it all inside the if-statement.

Note the double pair of parentheses. If you were to write it like this,

```
if (self = [super init]) {
  . . .
}
```

then Xcode complains. It is not sure whether you meant to make the assignment or whether you intended to compare the value of `self` to the return value of `[super init]`, in which case you should have used two equals signs:

```
if (self == [super init]) {
  . . .
}
```

That doesn't really make sense, as we definitely want to assign the result of `[super init]` to `self`, not compare the two. The extra pair of parentheses is used to make the intention clear to the compiler.

It doesn't really matter which approach you will use to write this, they are all equivalent:

```
self = [super init];
if (self) { . . . }

self = [super init];
if (self != nil) { . . . }

if ((self = [super init])) { . . . }

if ((self = [super init]) != nil) { . . . }
```

As long as you don't forget to do it!

It can happen that the call to `[super init]` returns `nil` in which case the initialization for the object's superclass failed and this object cannot be used. The object has already been destroyed at that point and there is nothing left to do but bail out. That's why we use the if-statement.

Fortunately, that doesn't happen a lot. An instance where this could happen is if your object tries to load an image but the image is not available and without it your object cannot function. Then you'd destroy the object and return `nil`. That is exactly what the `UIImage` object does if you give it the name of an image file that is not present in your application bundle.

If `self` is not `nil`, then the superclass was properly initialized and we can perform our own initialization. In `ChecklistItem`'s `init` method there really isn't a need to do any other initialization, which is why we haven't changed this method. However, in `initWithCoder` we initialize the object by reading the values from the `NSCoder` object and stuffing them into `ChecklistItem`'s properties.

Finally, you return `self`. That is necessary so the caller can assign the new object to a variable.

In summary, this is what an init method needs to do:

1. Call `[super init]` and assign the result to `self`.
2. Check whether `self` is `nil`. If so, then exit this method right away and return `nil` to the caller.
3. If `self` was not `nil`, do additional initialization if necessary. Usually this means you give properties and instance variables their initial values. By default, objects are `nil`, `int`s are 0 and `BOOL`s are `NO`. If you want to give these variables different initial values, then this is the place to do so.
4. Return `self` to the caller.

You don't always need to provide an init method. If your init method doesn't need to do anything — if there are no properties or instance variables to fill in — then you can leave it out completely and the compiler will provide one for you.

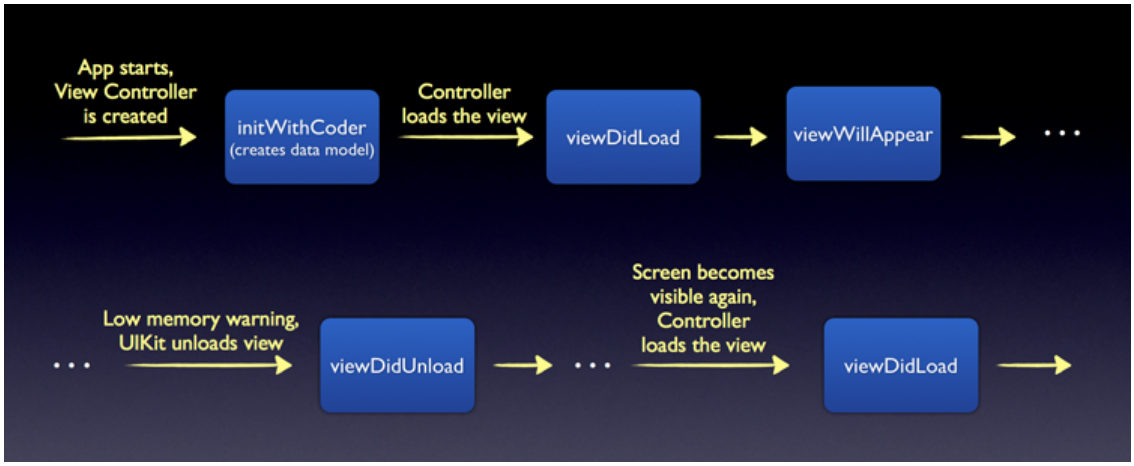
The implementation of `ChecklistItem` is complete, as it can now bring back to life objects that were serialized (or frozen) into the plist file. But we still have to write the code that will actually load this plist. That happens in `ChecklistsViewController`.

Until now we've done the initialization of our data model in `viewDidLoad`. That was convenient for the purposes of the tutorial but not always correct. `viewDidLoad` is the place where you should set up your views, i.e. the visible user elements from your app. Data model objects should be initialized earlier because typically the data model tends to live longer than the view.

When the iPhone runs low on available memory (and that happens more often than you'd think!), UIKit will unload the views of any view controller that is not currently visible. This could happen to our app too. The user may be working in the Add/Edit Item screen when the device gives out a "low memory" warning. To reclaim as much memory as possible, UIKit will then unload the `ChecklistsViewController`'s table view. That screen

is not visible at that point so it doesn't really need a live view object anyway.

The lifecycle of a view controller; viewDidLoad may be called more than once



Even when the view is (temporarily) unloaded, the data model needs to stick around. The Add/Edit Item screen will still send delegate messages to `ChecklistsViewController` and in response the controller needs to update the data model, even if the view is not present. We don't want to tie the lifecycle of the data model to the existence of the view, so we shouldn't create it in `viewDidLoad`. A better place for creating the data model is in the view controller's init method.

> The situation described above is how it works for iOS 5, but Apple changed the rules for iOS 6 so that views are no longer automatically unloaded in low-memory situations. That said, it's still wise not to create the data model in `viewDidLoad`, especially if you still want your apps to run on iOS 5.

A table view controller, like many objects, has more than one init method. There is:

- `initWithCoder`, for view controllers that are automatically loaded from a nib or Storyboard
- `initWithNibName`, for view controllers that you manually want to load from a nib
- `initWithStyle`, for table view controllers that you manually want to create without using a nib

Our view controller comes from a storyboard, so we'll use `initWithCoder` to create the data model and load the plist file. Yup, that's actually the same method we've just implemented in `ChecklistItem`. The `UITableViewController` object gets loaded and unfrozen

from the Storyboard file using the same `NSCoder` system that we used for our own files. If it's good enough for Storyboards then it's certainly good enough for us!

» Add the `initWithCoder` method above `viewDidLoad` in ChecklistsViewController.m:

ChecklistsViewController.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    [self loadChecklistItems];
  }
  return self;
}
```

Notice the same pattern: This is an init method so we call `super` and assign the result to `self`. If `self` is not `nil` we will do stuff, and finally we return a reference to `self`. The only difference is that we don't call `[super init]` but `[super initWithCoder]`. That ensures the rest of the view controller is properly unfrozen from the storyboard file.

» Add the `loadChecklistItems` method above `initWithCoder`:

ChecklistsViewController.m

```
- (void)loadChecklistItems
{
  NSString *path = [self dataFilePath];
  if ([[NSFileManager defaultManager] fileExistsAtPath:path])
  {
    NSData *data = [[NSData alloc] initWithContentsOfFile:path];
    NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc] ↵
        initForReadingWithData:data];
    items = [unarchiver decodeObjectForKey:@"ChecklistItems"];
    [unarchiver finishDecoding];
  }
  else
  {
    items = [[NSMutableArray alloc] initWithCapacity:20];
  }
}
```

Let's go through this. There are basically two paths that this method can take:

```
NSString *path = [self dataFilePath];
if ([[NSFileManager defaultManager] fileExistsAtPath:path])
{
  . . .
```

```
}
else
{
  items = [[NSMutableArray alloc] initWithCapacity:20];
}
```

We first put the results of `[self dataFilePath]` in a temporary variable named `path`. We will use the path name more than once in this method so having it available in a local variable instead of calling `dataFilePath` several times over is a small optimization.

Then we check whether the file actually exists and decide what happens based on that.

If there is no Checklists.plist then there are obviously no `ChecklistItem` objects for us to load. In that case, we go to the `else` section and create an empty `NSMutableArray`. That is what we used to do in `viewDidLoad`, so this shouldn't be too surprising. This is what happens when the app is started up for the very first time.

When we do have a Checklists.plist file, we don't have to make the array ourselves. We'll load the entire array and its contents from the Checklists.plist file:

```
NSData *data = [[NSData alloc] initWithContentsOfFile:path];
NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc] ↵
    initForReadingWithData:data];
items = [unarchiver decodeObjectForKey:@"ChecklistItems"];
[unarchiver finishDecoding];
```

This is essentially the reverse of what `saveChecklistItems` does. First we load the contents of the file into an `NSData` object. Then we create an `NSKeyedUnarchiver` object (note: this is an *un*archiver) and ask it to decode that data into our `items` array. This will create an `NSMutableArray` for us and populate it with exact copies of the `ChecklistItem` objects that were frozen into this file.

» You can now remove the code that created fake items from `viewDidLoad`:

ChecklistsViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];
}
```

There is currently nothing left to do in `viewDidLoad` (other than calling `super`), but we'll give it something new to do soon enough.

» Run the app and make some changes to the to-do items. Press Stop to terminate the app. Start it again and notice that your changes are still there.

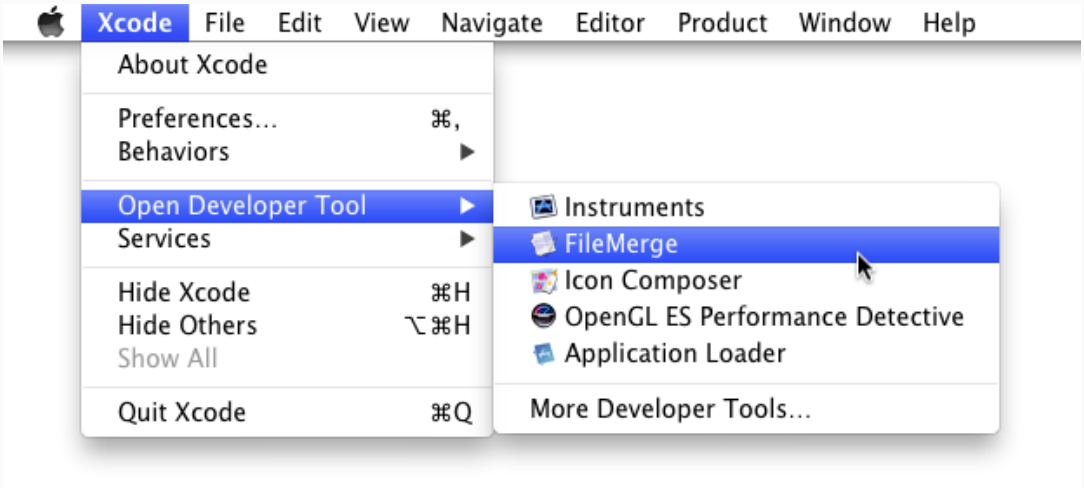
» Stop the app again. Go to the Finder window with the Documents folder and remove the Checklists.plist file. Run the app once more. You should now have an empty screen. Add an item and notice that the Checklists.plist file re-appears.

Awesome! We've written an app that not only lets you add and edit data, but that also persists that data between sessions. These techniques form the basis of many, many apps. Being able to use a navigation controller, show modal edit screens, and pass data around through delegates are essential iOS development skills.

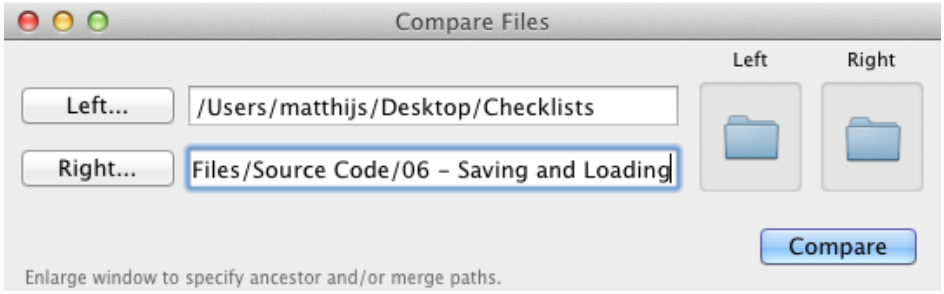
You can find the project files for the app up to this point under "06 - Saving and Loading" in the tutorial's Source Code folder.

Using FileMerge to compare files

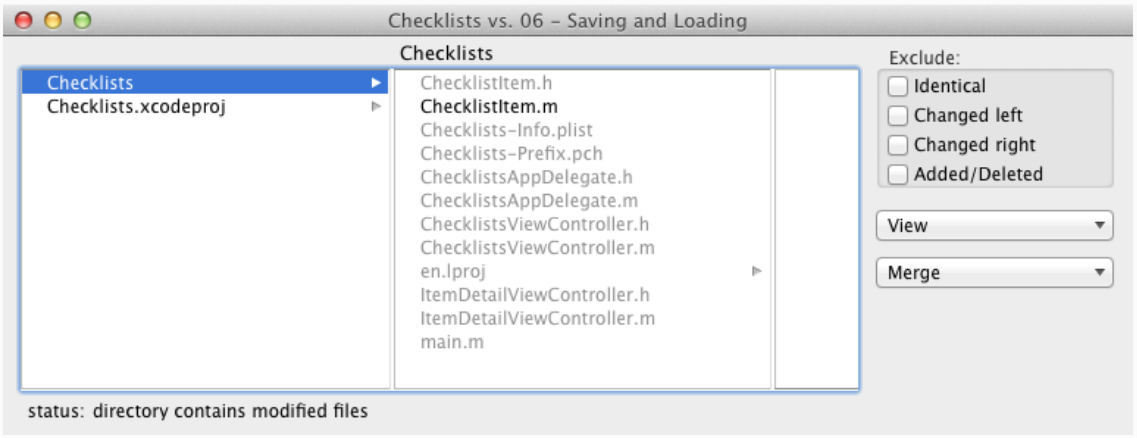
You can compare your own work with my version of the app using the FileMerge tool. With Xcode 4.2 you can find this tool in the /Developer/Applications folder, but as of Xcode 4.3 you have to open it from the Xcode menu:



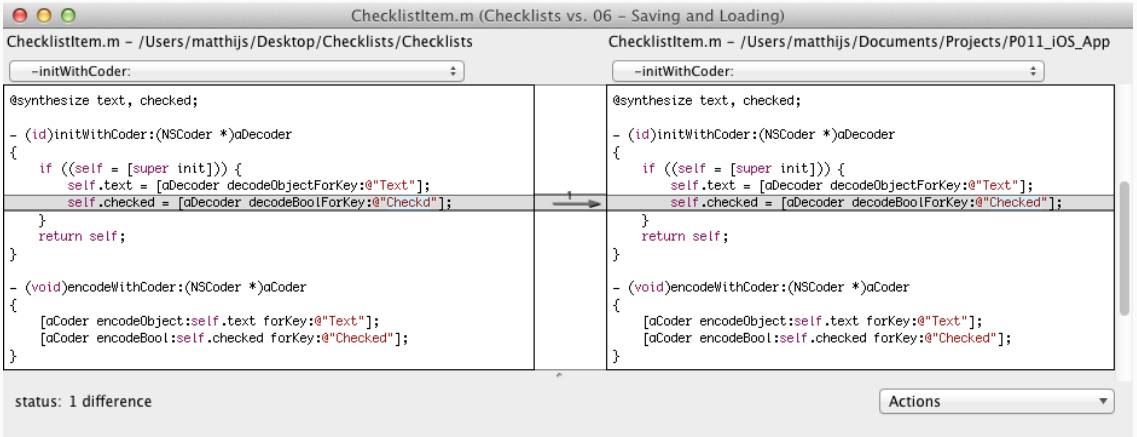
You give FileMerge two files or two folders to compare:



After working hard for a few seconds or so, FileMerge tells you what is different:



Double-click on a filename from the list and FileMerge shows the differences between the two files:



FileMerge is a wonderful tool for spotting the differences between two files or even entire folders. I use

it all the time! If something from the tutorials doesn't work as it should, then do a "diff" between your own files and the ones from the Source Code folder.

Just to make sure you get everything we've done, in the second part of this tutorial we will expand the app with some new features that more or less repeat what we just did. We'll also add cool stuff such as local notifications.